W3School Ruby 教程

wizardforcel

Published with GitBook

目錄

W3School Ruby教程

来源：Ruby教程

整理：飞龙

Ruby 基础

Ruby 简介

Ruby是一种纯粹的面向对象编程语言。它由日本的松本行弘（まつもとゆきひろ/Yukihiro Matsumoto）创建于1993年。

您可以在 www.ruby-lang.org 的 Ruby 邮件列表上找到松本行弘（まつもとゆきひろ/Yukihiro Matsumoto）的名字。在 Ruby 社区，松本也被称为马茨（Matz）。

Ruby 是"程序员的最佳朋友"。

Ruby 的特性与 Smalltalk、Perl 和 Python 类似。Perl、Python 和 Smalltalk 是脚本语言。Smalltalk 是一个真正的面向对象语言。Ruby，与 Smalltalk 一样，是一个完美的面向对象语言。使用 Ruby 的语法比使用 Smalltalk 的语法要容易得多。

Ruby 的特性

- Ruby 是开源的，在 Web 上免费提供，但需要一个许可证。
- Ruby 是一种通用的、解释的编程语言。
- Ruby 是一种真正的面向对象编程语言。
- Ruby 是一种类似于 Python 和 Perl 的服务器端脚本语言。
- Ruby 可以用来编写通用网关接口（CGI）脚本。
- Ruby 可以被嵌入到超文本标记语言（HTML）。
- Ruby 语法简单，这使得新的开发人员能够快速轻松地学习 Ruby。
- Ruby 与 C++ 和 Perl 等许多编程语言有着类似的语法。
- Ruby 可扩展性强，用 Ruby 编写的大程序易于维护。
- Ruby 可用于开发的 Internet 和 Intranet 应用程序。
- Ruby 可以安装在 Windows 和 POSIX 环境中。
- Ruby 支持许多 GUI 工具，比如 Tcl/Tk、GTK 和 OpenGL。
- Ruby 可以很容易地连接到 DB2、MySQL、Oracle 和 Sybase。
- Ruby 有丰富的内置函数，可以直接在 Ruby 脚本中使用。

您需要的工具

为了执行本教程中讨论的实例，您需要 RAM 至少为 2GB（推荐为 4GB）的 Intel Core i3 或 i5 的计算机。您还需要以下软件：

- Linux 或 Windows 95/98/2000/NT 或 Windows 7 操作系统
- Apache 1.3.19-5 Web 服务器
- Internet Explorer 5.0 或以上的 Web 浏览器

- Ruby 1.8.5

本教程将介绍如何使用 Ruby 创建 GUI、网络和 Web 应用程序。另外还会讨论如何扩展和嵌入 Ruby 应用程序。

接下来将学习什么？

下一章将向您介绍从哪里可以获取 Ruby 及其文档。最后，它会指示您如何安装 Ruby，并配置环境为开发 Ruby 应用程序做准备。

Ruby 环境

本地环境设置

如果您想要设置 Ruby 编程语言的环境，请阅读本章节的内容。本章将向您讲解与环境设置有关的所有重要的主题。建议先学习下面几个主题，然后再进一步深入学习其他主题：

- Linux/Unix 上的 Ruby 安装：如果您想要在 Linux/Unix 上配置开发环境，那么请查看本章节的内容。
- Windows 上的 Ruby 安装：如果您想要在 Windows 上配置开发环境，那么请查看本章节的内容。
- Ruby 命令行选项：本章节列出了所有的命令行选项，您可以和 Ruby 解释器一起使用这些命令行选项。
- Ruby 环境变量：本章节列出了所有重要的环境变量列表，设置这些环境变量以便让 Ruby 解释器工作。

流行的 Ruby 编辑器

为了编写 Ruby 程序，您需要一个编辑器：

- 如果您是在 Windows 上进行编写，那么您可以使用任何简单的文本编辑器，比如 Notepad 或 Edit plus。
- VIM（Vi IMproved）是一个简单的文本编辑器，几乎在所有的 Unix 上都是可用的，现在也能在 Windows 上使用。另外，您还可以使用您喜欢的 vi 编辑器来编写 Ruby 程序。
- RubyWin 是一个针对 Windows 的 Ruby 集成开发环境（IDE）。
- Ruby Development Environment （RDE） 对于 Windows 用户来说，也是一个很好的集成开发环境（IDE）。

交互式 Ruby（IRb）

交互式 Ruby（IRb）为体验提供了一个 shell。在 IRb shell 内，您可以逐行立即查看解释结果。

这个工具会随着 Ruby 的安装自动带有，所以您不需要做其他额外的事情，IRb 即可正常工作。

只需要在命令提示符中键入 **irb**，一个交互式 Ruby Session 将会开始，如下所示：

```
$irb
irb 0.6.1(99/09/16)
irb(main):001:0> def hello
irb(main):002:1> out = "Hello World"
irb(main):003:1> puts out
irb(main):004:1> end
nil
irb(main):005:0> hello
Hello World
nil
irb(main):006:0>
```

这里您可以先不用关心上面命令的执行内容，我们将在后续的章节中向您讲解。

接下来将学习什么？

假设现在您已经设置好 Ruby 环境，且已经做好编写第一个 Ruby 程序的准备。下一章我们将向您讲解如何编写 Ruby 程序。

Ruby 安装 - Unix

下面列出了在 Unix 机器上安装 Ruby 的步骤。

注意：在安装之前，请确保您有 root 权限。

- 下载最新版的 Ruby 压缩文件。请点击这里下载。
- 下载 Ruby 之后，解压到新创建的目录下：

```
$ tar -xvzf ruby-1.6.7.tgz
$ cd ruby-1.6.7
```

- 现在，配置并编译源代码，如下所示：

```
$ ./configure
$ make
```

- 最后，安装 Ruby 解释器，如下所示：

```
$ su -l root # 使用root用户
$ make install
$ exit       # 切换回普通用户
```

- 安装后，通过在命令行中输入以下命令来确保一切工作正常：

```
$ruby -v
ruby 1.6.7 (2002-06-04) [i386-netbsd]
```

- 如果一切工作正常，将会输出所安装的 Ruby 解释器的版本，如上所示。如果您安装了其他版本，则会显示其他不同的版本。

使用 yum 安装 Ruby

如果您的计算机已经连接到 Internet，那么最简单的安装 Ruby 的方式是使用 **yum**。在命令提示符中输入以下的命令，即可在您的计算机上安装 Ruby。

```
$ yum install ruby
```

Ruby 安装 - Windows

下面列出了在 Windows 机器上安装 Ruby 的步骤。

注意：在安装时，您可能有不同的可用版本。

- 下载最新版的 Ruby 压缩文件。请点击这里下载。
- 下载 Ruby 之后，解压到新创建的目录下：
- 双击 Ruby1.6.7.exe 文件，启动 Ruby 安装向导。
- 点击 Next，继续向导的 Important Information 页面，直到 Ruby 安装程序完成 Ruby 安装为止。

如果您的安装没有适当地配置环境变量，接下来您可能需要进行环境变量的配置。

- 如果您使用的是 Windows 9x，那么请在您的 c:\autoexec.bat 中添加：set PATH="D:(ruby 安装目录)\bin;%PATH%"
- Windows NT/2000 用户需要修改注册表。
    - 点击控制面板|系统性能|环境变量。
    - 在系统变量下，选择 Path，并点击 EDIT。
    - 在变量值列表的末尾添加 Ruby 目录，并点击 OK。
    - 在系统变量下，选择 PATHEXT，并点击 EDIT。
    - 添加 .RB 和 .RBW 到变量值列表中，并点击 OK。
- 安装后，通过在命令行中输入以下命令来确保一切工作正常：

```
$ruby -v
ruby 1.6.7
```

- 如果一切工作正常，将会输出所安装的 Ruby 解释器的版本，如上所示。如果您安装了其他版本，则会显示其他不同的版本。

Ruby 命令行选项

Ruby 一般是从命令行运行，方式如下：

```
$ ruby [ options ] [.] [ programfile ] [ arguments ... ]
```

解释器可以通过下列选项被调用，来控制解释器的环境和行为。

选项	描述
-a	与 -n 或 -p 一起使用时，可以打开自动拆分模式(auto split mode)。请查看 -n 和 -p 选项。
-c	只检查语法，不执行程序。
-C dir	在执行前改变目录（等价于 -X）。
-d	启用调试模式（等价于 -debug）。
-F pat	指定 pat 作为默认的分离模式（$;）。
-e prog	指定 prog 作为程序在命令行中执行。可以指定多个 -e 选项，用来执行多个程序。
-h	显示命令行选项的一个概览。
-i [ext]	把文件内容重写为程序输出。原始文件会被加上扩展名 ext 保存下来。如果未指定 ext，原始文件会被删除。
-I dir	添加 dir 作为加载库的目录。
-K [kcode]	指定多字节字符集编码。e 或 E 对应 EUC（extended Unix code），s 或 S 对应 SJIS（Shift-JIS），u 或 U 对应 UTF-8，a、A、n 或 N 对应 ASCII。
-l	启用自动行尾处理。从输入行取消一个换行符，并向输出行追加一个换行符。
-n	把代码放置在一个输入循环中（就像在 while gets; ... end 中一样）。
-0[octal]	设置默认的记录分隔符（$/）为八进制。如果未指定 octal 则默认为 \0。
-p	把代码放置在一个输入循环中。在每次迭代后输出变量 $_ 的值。
-r lib	使用 *require* 来加载 *lib* 作为执行前的库。
-s	解读程序名称和文件名参数之间的匹配模式 -xxx 的任何参数作为开关，并定义相应的变量。
-T [level]	设置安全级别，执行不纯度测试（如果未指定 level，则默认值为 1）。
-v	显示版本，并启用冗余模式。
-w	启用冗余模式。如果未指定程序文件，则从 STDIN 读取。

-x [dir]	删除 #!ruby 行之前的文本。如果指定了 *dir*，则把目录改变为 *dir*。
-X dir	在执行前改变目录（等价于 -C）。
-y	启用解析器调试模式。
--copyright	显示版权声明。
--debug	启用调试模式（等价于 -d）。
--help	显示命令行选项的一个概览（等价于 -h）。
--version	显示版本。
--verbose	启用冗余模式（等价于 -v）。设置 $VERBOSE 为 true。
--yydebug	启用解析器调试模式（等价于 -y）。

单字符的命令行选项可以组合使用。下面两行表达了同样的意思：

```
$ruby -ne 'print if /Ruby/' /usr/share/bin

$ruby -n -e 'print if /Ruby/' /usr/share/bin
```

Ruby 环境变量

Ruby 解释器使用下列环境变量来控制它的行为。ENV 对象包含了所有当前设置的环境变量列表。

变量	描述
DLN_LIBRARY_PATH	动态加载模块搜索的路径。
HOME	当没有参数传递给 Dir::chdir 时，要移动到的目录。也用于 File::expand_path 来扩展 "~"。
LOGDIR	当没有参数传递给 Dir::chdir 且未设置环境变量 HOME 时，要移动到的目录。
PATH	执行子进程的搜索路径，以及在指定 -S 选项后，Ruby 程序的搜索路径。每个路径用冒号分隔（在 DOS 和 Windows 中用分号分隔）。
RUBYLIB	库的搜索路径。每个路径用冒号分隔（在 DOS 和 Windows 中用分号分隔）。
RUBYLIB_PREFIX	用于修改 RUBYLIB 搜索路径，通过使用格式 path1;path2 或 path1path2，把库的前缀 path1 替换为 path2。
RUBYOPT	传给 Ruby 解释器的命令行选项。在 taint 模式时被忽略（其中，$SAFE 大于 0）。
RUBYPATH	指定 -S 选项后，Ruby 程序的搜索路径。优先级高于 PATH。在 taint 模式时被忽略（其中，$SAFE 大于 0）。
RUBYSHELL	指定执行命令时所使用的 shell。如果未设置该环境变量，则使用 SHELL 或 COMSPEC。

对于 Unix，使用 **env** 命令来查看所有环境变量的列表。

```
HOSTNAME=ip-72-167-112-17.ip.secureserver.net
RUBYPATH=/usr/bin
SHELL=/bin/bash
TERM=xterm
HISTSIZE=1000
SSH_CLIENT=122.169.131.179 1742 22
SSH_TTY=/dev/pts/1
USER=amrood
JRE_HOME=/usr/java/jdk/jre
J2RE_HOME=/usr/java/jdk/jre
PATH=/usr/local/bin:/bin:/usr/bin:/home/guest/bin
MAIL=/var/spool/mail/guest
PWD=/home/amrood
INPUTRC=/etc/inputrc
JAVA_HOME=/usr/java/jdk
LANG=C
HOME=/root
SHLVL=2
JDK_HOME=/usr/java/jdk
LOGDIR=/usr/log/ruby
LOGNAME=amrood
SSH_CONNECTION=122.169.131.179 1742 72.167.112.17 22
LESSOPEN=|/usr/bin/lesspipe.sh %s
RUBYLIB=/usr/lib/ruby
G_BROKEN_FILENAMES=1
_=/bin/env
```

Ruby 语法

让我们编写一个简单的 Ruby 程序。所有的 Ruby 文件扩展名都是 **.rb**。所以，把下面的源代码放在 test.rb 文件中。

```
#!/usr/bin/ruby -w

puts "Hello, Ruby!";
```

在这里，假设您的 /usr/bin 目录下已经有可用的 Ruby 解释器。现在，尝试运行这个程序，如下所示：

```
$ ruby test.rb
```

这将会产生下面的结果：

```
Hello, Ruby!
```

您已经看到了一个简单的 Ruby 程序，现在让我们看看一些 Ruby 语法相关的基本概念：

Ruby 程序中的空白

在 Ruby 代码中的空白字符，如空格和制表符一般会被忽略，除非当它们出现在字符串中时才不会被忽略。然而，有时候它们用于解释模棱两可的语句。当启用 -w 选项时，这种解释会产生警告。

实例：

```
a + b 被解释为 a+b （这是一个局部变量）
a  +b 被解释为 a(+b) （这是一个方法调用）
```

Ruby 程序中的行尾

Ruby 把分号和换行符解释为语句的结尾。但是，如果 Ruby 在行尾遇到运算符，比如 +、- 或反斜杠，它们表示一个语句的延续。

Ruby 标识符

标识符是变量、常量和方法的名称。Ruby 标识符是大小写敏感的。这意味着 Ram 和 RAM 在 Ruby 中是两个不同的标识符。

Ruby 标识符的名称可以包含字母、数字和下划线字符（ _ ）。

保留字

下表列出了 Ruby 中的保留字。这些保留字不能作为常量或变量的名称。但是，它们可以作为方法名。

BEGIN	do	next	then
END	else	nil	true
alias	elsif	not	undef
and	end	or	unless
begin	ensure	redo	until
break	false	rescue	when
case	for	retry	while
class	if	return	while
def	in	self	__FILE__
defined?	module	super	__LINE__

Ruby 中的 Here Document

"Here Document" 是指建立多行字符串。在 << 之后，您可以指定一个字符串或标识符来终止字符串，且当前行之后直到终止符为止的所有行是字符串的值。

如果终止符用引号括起，引号的类型决定了面向行的字符串类型。请注意<< 和终止符之间必须没有空格。

下面是不同的实例：

```
#!/usr/bin/ruby -w

print <<EOF
    This is the first way of creating
    here document ie. multiple line string.
EOF

print <<"EOF";                # 与上面相同
    This is the second way of creating
    here document ie. multiple line string.
EOF

print <<`EOC`                 # 执行命令
    echo hi there
    echo lo there
EOC

print <<"foo", <<"bar"          # 您可以把它们进行堆叠
    I said foo.
foo
    I said bar.
bar
```

这将产生以下结果：

```
    This is the first way of creating
    her document ie. multiple line string.
    This is the second way of creating
    her document ie. multiple line string.
hi there
lo there
        I said foo.
        I said bar.
```

Ruby *BEGIN* 语句

语法

```
BEGIN {
   code
}
```

声明 *code* 会在程序运行之前被调用。

实例

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

BEGIN {
   puts "Initializing Ruby Program"
}
```

这将产生以下结果：

```
Initializing Ruby Program
This is main Ruby Program
```

Ruby *END* 语句

语法

```
END {
   code
}
```

声明 *code* 会在程序的结尾被调用。

实例

```
#!/usr/bin/ruby

puts "This is main Ruby Program"

END {
   puts "Terminating Ruby Program"
}
BEGIN {
   puts "Initializing Ruby Program"
}
```

这将产生以下结果：

```
Initializing Ruby Program
This is main Ruby Program
Terminating Ruby Program
```

Ruby 注释

注释会对 Ruby 解释器隐藏一行，或者一行的一部分，或者若干行。您可以在行首使用字符（#）：

```
# 我是注释，请忽略我。
```

或者，注释可以跟着语句或表达式的同一行的后面：

```
name = "Madisetti" # 这也是注释
```

您可以注释多行，如下所示：

```
# 这是注释。
# 这也是注释。
# 这也是注释。
# 这还是注释。
```

下面是另一种形式。这种块注释会对解释器隐藏 =begin/=end 之间的行：

```
=begin
这是注释。
这也是注释。
这也是注释。
这还是注释。
=end
```

Ruby 数据类型

本章节我们将为大家介绍 Ruby 的基本数据类型。

Ruby支持的数据类型包括基本的Number、String、Ranges、Symbols，以及true、false和nil这几个特殊值，同时还有两种重要的数据结构——Array和Hash。

数值类型(Number)

1、整型(Integer)

整型分两种，如果在31位以内（四字节），那为Fixnum实例。如果超过，即为Bignum实例。

整数范围从 -2³⁰ 到 2³⁰⁻¹ 或 -2⁶² 到 2⁶²⁻¹。在这个范围内的整数是类 *Fixnum* 的对象，在这个范围外的整数存储在类 *Bignum* 的对象中。

您可以在整数前使用一个可选的前导符号，一个可选的基础指标（0 对应 octal，0x 对应 hex，0b 对应 binary），后跟一串数字。下划线字符在数字字符串中被忽略。

您可以获取一个 ASCII 字符或一个用问号标记的转义序列的整数值。

实例

```
123                  # Fixnum 十进制
1_234                # Fixnum 带有下划线的十进制
-500                 # 负的 Fixnum
0377                 # 八进制
0xff                 # 十六进制
0b1011               # 二进制
?a                   # 'a' 的字符编码
?\n                  # 换行符（0x0a）的编码
12345678901234567890 # Bignum
```

```
#整型 Integer 以下是一些整型字面量
#字面量（literal）：代码中能见到的值，数值，bool值，字符串等都叫字面量
#如以下的0,1_000_000,0xa等
a1=0

#带千分符的整型
a2=1_000_000

#其它进制的表示
a3=0xa
puts a1,a2
puts a3

#puts print 都是向控制台打印字符，其中puts带回车换行符
=begin
这是注释，称作：嵌入式文档注释
类似C#中的/**/
=end
```

浮点型

Ruby 支持浮点数。它们是带有小数的数字。浮点数是类 *Float* 的对象，且可以是下列中任意一个。

实例

```
123.4                # 浮点值
1.0e6                # 科学记数法
4E20                 # 不是必需的
4e+20                # 指数前的符号
```

```
#浮点型
f1=0.0
f2=2.1
f3=1000000.1
puts f3
```

算术操作

加减乘除操作符：+-/；指数操作符为*

指数不必是整数，例如

```
#指数算术
puts 2**(1/4)#1与4的商为0，然后2的0次方为1
puts 16**(1/4.0)#1与4.0的商为0.25（四分之一），然后开四次方根
```

字符串类型

Ruby 字符串简单地说是一个 8 位字节序列，它们是类 String 的对象。

双引号标记的字符串允许替换和使用反斜线符号，单引号标记的字符串不允许替换，且只允许使用 \ 和 \' 两个反斜线符号。

实例

```
#!/usr/bin/ruby -w

puts 'escape using "\\"';
puts 'That\'s right';
```

这将产生以下结果：

```
escape using "\"
That's right
```

您可以使用序列 **#{ expr }** 替换任意 Ruby 表达式的值为一个字符串。在这里，expr 可以是任意的 Ruby 表达式。

```
#!/usr/bin/ruby -w

puts "Multiplication Value : #{24*60*60}";
```

这将产生以下结果：

```
Multiplication Value : 86400
```

```
#!/usr/bin/ruby -w

name="Ruby"
puts name
puts "#{name+",ok"}"
```

输出结果为：

```
Ruby
Ruby,ok
```

反斜线符号

下表列出了 Ruby 支持的反斜线符号：

符号	表示的字符
\n	换行符 (0x0a)
\r	回车符 (0x0d)
\f	换页符 (0x0c)
\b	退格键 (0x08)
\a	报警符 Bell (0x07)
\e	转义符 (0x1b)
\s	空格符 (0x20)
\nnn	八进制表示法 (n 是 0-7)
\xnn	十六进制表示法 (n 是 0-9、a-f 或 A-F)
\cx, \C-x	Control-x
\M-x	Meta-x (c | 0x80)
\M-\C-x	Meta-Control-x
\x	字符 x

如需了解更多有关 Ruby 字符串的细节，请查看 Ruby 字符串（String）。

数组

数组字面量通过[]中以逗号分隔定义，且支持range定义。

- （1）数组通过[]索引访问
- （2）通过赋值操作插入、删除、替换元素
- （3）通过+，－号进行合并和删除元素，且集合做为新集合出现
- （4）通过<<号向原数据追加元素
- （5）通过*号重复数组元素
- （6）通过｜和&符号做并集和交集操作（注意顺序）

实例：

```
#!/usr/bin/ruby

ary = [  "fred", 10, 3.14, "This is a string", "last element", ]
ary.each do |i|
   puts i
end
```

这将产生以下结果：

```
fred
10
3.14
This is a string
last element
```

如需了解更多有关 Ruby 数组的细节，请查看 Ruby 数组（Array）。

哈希类型

Ruby 哈希是在大括号内放置一系列键/值对，键和值之间使用逗号和序列 => 分隔。尾部的逗号会被忽略。

实例

```
#!/usr/bin/ruby

hsh = colors = { "red" => 0xf00, "green" => 0x0f0, "blue" => 0x00f }
hsh.each do |key, value|
   print key, " is ", value, "\n"
end
```

这将产生以下结果：

```
green is 240
red is 3840
blue is 15
```

如需了解更多有关 Ruby 哈希的细节，请查看 Ruby 哈希（Hash）。

范围类型

一个范围表示一个区间。

范围是通过设置一个开始值和一个结束值来表示。范围可使用 s..e 和 s...e 来构造，或者通过 Range.new 来构造。

使用 .. 构造的范围从开始值运行到结束值（包含结束值）。使用 ... 构造的范围从开始值运行到结束值（不包含结束值）。当作为一个迭代器使用时，范围会返回序列中的每个值。

范围 (1..5) 意味着它包含值 1, 2, 3, 4, 5，范围 (1...5) 意味着它包含值 1, 2, 3, 4 。

实例

```
#!/usr/bin/ruby

(10..15).each do |n|
   print n, ' '
end
```

这将产生以下结果：

```
10 11 12 13 14 15
```

如需了解更多有关 Ruby 范围的细节，请查看 Ruby 范围（Range）。

Ruby 类和对象

Ruby 是一种完美的面向对象编程语言。面向对象编程语言的特性包括：

- 数据封装
- 数据抽象
- 多态性
- 继承

这些特性将在 面向对象的 Ruby 中进行讨论。

一个面向对象的程序，涉及到的类和对象。类是个别对象创建的蓝图。在面向对象的术语中，您的自行车是自行车类的一个实例。

以车辆为例，它包括车轮（wheels）、马力（horsepower）、燃油或燃气罐容量（fuel or gas tank capacity）。这些属性形成了车辆（Vehicle）类的数据成员。借助这些属性您能把一个车辆从其他车辆中区分出来。

车辆也能包含特定的函数，比如暂停（halting）、驾驶（driving）、超速（speeding）。这些函数形成了车辆（Vehicle）类的数据成员。因此，您可以定义类为属性和函数的组合。

类 Vehicle 的定义如下：

```
Class Vehicle
{
   Number no_of_wheels
   Number horsepower
   Characters type_of_tank
   Number Capacity
   Function speeding
   {
   }
   Function driving
   {
   }
   Function halting
   {
   }
}
```

通过给这些数据成员分配不同的值，您可以创建类 Vehicle 的不同实例。例如，一架飞机有三个轮子，马力 1,000，燃油罐容量为 100 升。以同样的方式，一辆汽车有四个轮子，马力 200，煤气罐容量为 25 升。

在 Ruby 中定义类

为了使用 Ruby 实现面向对象编程，您需要先学习如何在 Ruby 中创建对象和类。

在 Ruby 中，类总是以关键字 *class* 开始，后跟类的名称。类名的首字母应该大写。类 *Customer* 如下所示：

```
class Customer
end
```

您可以使用关键字 *end* 终止一个类。类 中的所有数据成员都是介于类定义和 *end* 关键字之间。

Ruby 类中的变量

Ruby 提供了四种类型的变量：

- 局部变量：局部变量是在方法中定义的变量。局部变量在方法外是不可用的。在后续的章节中，您将看到有关方法的更多细节。局部变量以小写字母或 _ 开始。
- 实例变量：实例变量可以跨任何特定的实例或对象中的方法使用。这意味着，实例变量可以从对象到对象的改变。实例变量在变量名之前放置符号（@）。
- 类变量：类变量可以跨不同的对象使用。类变量属于类，且是类的一个属性。类变量在变量名之前放置符号（@@）。
- 全局变量：类变量不能跨类使用。如果您想要有一个可以跨类使用的变量，您需要定义全局变量。全局变量总是以美元符号（$）开始。

实例

使用类变量 @@no_of_customers，您可以判断被创建的对象数量，这样可以确定客户数量。

```
class Customer
   @@no_of_customers=0
end
```

在 Ruby 中使用 *new* 方法创建对象

对象是类的实例。现在您将学习如何在 Ruby 中创建类的对象。在 Ruby 中，您可以使用类的方法 *new* 创建对象。

方法 *new* 是一种独特的方法，在 Ruby 库中预定义。new 方法属于类方法。

下面的实例创建了类 Customer 的两个对象 cust1 和 cust2：

```
cust1 = Customer. new
cust2 = Customer. new
```

在这里，cust1 和 cust2 是两个对象的名称。对象名称后跟着等号（=），等号后跟着类名，然后是点运算符和关键字 *new*。

自定义方法来创建 Ruby 对象

您可以给方法 *new* 传递参数，这些参数可用于初始化类变量。

当您想要声明带参数的 *new* 方法时，您需要在创建类的同时声明方法 *initialize*。

initialize 方法是一种特殊类型的方法，将在调用带参数的类的 *new* 方法时执行。

下面的实例创建了 initialize 方法：

```
class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
end
```

在本实例中，您可以声明带有 **id**、**name**、**addr** 作为局部变量的 *initialize*方法。在这里，*def* 和 *end* 用于定义 Ruby 方法 *initialize*。在后续的章节中，您将学习有关方法的更多细节。

在 *initialize* 方法中，把这些局部变量的值传给实例变量 @cust_id、@cust_name 和 @cust_addr。在这里，局部变量的值是随着 new 方法进行传递的。

现在，您可以创建对象，如下所示：

```
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
```

Ruby 类中的成员函数

在 Ruby 中，函数被称为方法。类中的每个方法是以关键字 *def* 开始，后跟方法名。

方法名总是以小写字母开头。在 Ruby 中，您可以使用关键字 *end* 来结束一个方法。

下面的实例定义了一个 Ruby 方法：

```
class Sample
   def function
      statement 1
      statement 2
   end
end
```

在这里，*statement 1* 和 *statement 2* 是类 Sample 内的方法 *function* 的主体的组成部分。这些语句可以是任何有效的 Ruby 语句。例如，我们可以使用方法 *puts* 来输出 *Hello Ruby*，如下所示：

```
class Sample
   def hello
      puts "Hello Ruby!"
   end
end
```

下面的实例将创建类 Sample 的一个对象，并调用 *hello* 方法：

```
#!/usr/bin/ruby

class Sample
   def hello
      puts "Hello Ruby!"
   end
end

# 使用上面的类来创建对象
object = Sample. new
object.hello
```

这将会产生下面的结果：

```
Hello Ruby!
```

简单的案例研究

如果您想要做更多有关类和对象的练习，这里有一个案例研究：

Ruby 类案例

Ruby 类案例

下面将创建一个名为 Customer 的 Ruby 类，您将声明两个方法：

- *display_details*：该方法用于显示客户的详细信息。
- *total_no_of_customers*：该方法用于显示在系统中创建的客户总数量。

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
    def total_no_of_customers()
       @@no_of_customers += 1
       puts "Total number of customers: #@@no_of_customers"
    end
end
```

display_details 方法包含了三个 puts 语句，显示了客户 ID、客户名字和客户地址。其中，puts 语句：

```
puts "Customer id #@cust_id"
```

将在一个单行上显示文本 Customer id，后跟变量 @cust_id 的值。

当您想要在一个单行上显示实例变量的文本和值时，您需要在 puts 语句的变量名前面放置符号（#）。文本和带有符号（#）的实例变量应使用双引号标记。

第二个方法，total*no_of_customers*，包含了类变量 @@*no_of_customers*。表达式 @@*no_of* customers+=1 在每次调用方法 total_no_of_customers 时，把变量 no_of_customers 加 1。通过这种方式，您将得到类变量中的客户总数量。

现在创建两个客户，如下所示：

```
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")
```

在这里，我们创建了 Customer 类的两个对象，cust1 和 cust2，并向 new 方法传递必要的参数。当 initialize 方法被调用时，对象的必要属性被初始化。

一旦对象被创建，您需要使用两个对象来调用类的方法。如果您想要调用方法或任何数据成员，您可以编写代码，如下所示：

```
cust1.display_details()
cust1.total_no_of_customers()
```

对象名称后总是跟着一个点号，接着是方法名称或数据成员。我们已经看到如何使用 cust1 对象调用两个方法。使用 cust2 对象，您也可以调用两个方法，如下所示：

```
cust2.display_details()
cust2.total_no_of_customers()
```

保存并执行代码

现在，把所有的源代码放在 main.rb 文件中，如下所示：

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
   end
   def total_no_of_customers()
      @@no_of_customers += 1
      puts "Total number of customers: #@@no_of_customers"
   end
end

# 创建对象
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# 调用方法
cust1.display_details()
cust1.total_no_of_customers()
cust2.display_details()
cust2.total_no_of_customers()
```

接着，运行程序，如下所示：

```
$ ruby main.rb
```

这将产生以下结果：

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Total number of customers: 1
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
Total number of customers: 2
```

Ruby 变量

变量是持有可被任何程序使用的任何数据的存储位置。

Ruby 支持五种类型的变量。您已经在前面的章节中大概了解了这些变量，本章节将为您详细讲解这五种类型的变量。

Ruby 全局变量

全局变量以 $ 开头。未初始化的全局变量的值为 *nil*，在使用 -w 选项后，会产生警告。

给全局变量赋值会改变全局状态，所以不建议使用全局变量。

下面的实例显示了全局变量的用法。

```
#!/usr/bin/ruby

$global_variable = 10
class Class1
  def print_global
     puts "Global variable in Class1 is #$global_variable"
  end
end
class Class2
  def print_global
     puts "Global variable in Class2 is #$global_variable"
  end
end

class1obj = Class1.new
class1obj.print_global
class2obj = Class2.new
class2obj.print_global
```

在这里，$global_variable 是全局变量。这将产生以下结果：

注意：在 Ruby 中，您可以通过在变量或常量前面放置 # 字符，来访问任何变量或常量的值。

```
Global variable in Class1 is 10
Global variable in Class2 is 10
```

Ruby 实例变量

实例变量以 @ 开头。未初始化的实例变量的值为 *nil*，在使用 -w 选项后，会产生警告。

下面的实例显示了实例变量的用法。

```
#!/usr/bin/ruby

class Customer
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
end

# 创建对象
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# 调用方法
cust1.display_details()
cust2.display_details()
```

在这里，@cust_id、@cust_name 和 @cust_addr 是实例变量。这将产生以下结果：

```
Customer id 1
Customer name John
Customer address Wisdom Apartments, Ludhiya
Customer id 2
Customer name Poul
Customer address New Empire road, Khandala
```

Ruby 类变量

类变量以 @@ 开头，且必须初始化后才能在方法定义中使用。

引用一个未初始化的类变量会产生错误。类变量在定义它的类或模块的子类或子模块中可共享使用。

在使用 -w 选项后，重载类变量会产生警告。

下面的实例显示了类变量的用法。

```
#!/usr/bin/ruby

class Customer
   @@no_of_customers=0
   def initialize(id, name, addr)
      @cust_id=id
      @cust_name=name
      @cust_addr=addr
   end
   def display_details()
      puts "Customer id #@cust_id"
      puts "Customer name #@cust_name"
      puts "Customer address #@cust_addr"
    end
    def total_no_of_customers()
       @@no_of_customers += 1
       puts "Total number of customers: #@@no_of_customers"
    end
end

# 创建对象
cust1=Customer.new("1", "John", "Wisdom Apartments, Ludhiya")
cust2=Customer.new("2", "Poul", "New Empire road, Khandala")

# 调用方法
cust1.total_no_of_customers()
cust2.total_no_of_customers()
```

在这里，@@no_of_customers 是类变量。这将产生以下结果：

```
Total number of customers: 1
Total number of customers: 2
```

Ruby 局部变量

局部变量以小写字母或下划线 _ 开头。局部变量的作用域从 class、module、def 或 do 到相对应的结尾或者从左大括号到右大括号 {}。

当调用一个未初始化的局部变量时，它被解释为调用一个不带参数的方法。

对未初始化的局部变量赋值也可以当作是变量声明。变量会一直存在，直到当前域结束为止。局部变量的生命周期在 Ruby 解析程序时确定。

在上面的实例中，局部变量是 id、name 和 addr。

Ruby 常量

常量以大写字母开头。定义在类或模块内的常量可以从类或模块的内部访问，定义在类或模块外的常量可以被全局访问。

常量不能定义在方法内。引用一个未初始化的常量会产生错误。对已经初始化的常量赋值会产生警告。

```
#!/usr/bin/ruby

class Example
   VAR1 = 100
   VAR2 = 200
   def show
       puts "Value of first Constant is #{VAR1}"
       puts "Value of second Constant is #{VAR2}"
   end
end

# 创建对象
object=Example.new()
object.show
```

在这里，VAR1 和 VAR2 是常量。这将产生以下结果：

```
Value of first Constant is 100
Value of second Constant is 200
```

Ruby 伪变量

它们是特殊的变量，有着局部变量的外观，但行为却像常量。您不能给这些变量赋任何值。

- **self:** 当前方法的接收器对象。
- **true:** 代表 true 的值。
- **false:** 代表 false 的值。
- **nil:** 代表 undefined 的值。
- **FILE:** 当前源文件的名称。
- **LINE:** 当前行在源文件中的编号。

Ruby 运算符

Ruby 支持一套丰富的运算符。大多数运算符实际上是方法调用。例如，a + b 被解释为 a.+(b)，其中指向变量 a 的 + 方法被调用，b 作为方法调用的参数。

对于每个运算符（+ - / % * & | ^ << >> && ||），都有一个相对应的缩写赋值运算符（+= -= 等等）。

Ruby 算术运算符

假设变量 a 的值为 10，变量 b 的值为 20，那么：

运算符	描述	实例
+	加法 - 把运算符两边的操作数相加	a + b 将得到 30
-	减法 - 把左操作数减去右操作数	a - b 将得到 -10
*	乘法 - 把运算符两边的操作数相乘	a * b 将得到 200
/	除法 - 把左操作数除以右操作数	b / a 将得到 2
%	求模 - 把左操作数除以右操作数，返回余数	b % a 将得到 0
**	指数 - 执行指数计算	a**b 将得到 10 的 20 次方

Ruby 比较运算符

假设变量 a 的值为 10，变量 b 的值为 20，那么：

运算符	描述	实例
==	检查两个操作数的值是否相等，如果相等则条件为真。	(a == b) 不为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。	(a != b) 为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。	(a > b) 不为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。	(a < b) 为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。	(a >= b) 不为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。	(a <= b) 为真。
<=>	联合比较运算符。如果第一个操作数等于第二个操作数则返回 0，如果第一个操作数大于第二个操作数则返回 1，如果第一个操作数小于第二个操作数则返回 -1。	(a <=> b) 返回 -1。
===	用于测试 *case* 语句的 when 子句内的相等。	(1...10) === 5 返回 true。
.eql?	如果接收器和参数具有相同的类型和相等的值，则返回 true。	1 == 1.0 返回 true，但是 1**.eql?**(1.0) 返回 false。
equal?	如果接收器和参数具有相同的对象 id，则返回 true。	如果 aObj 是 bObj 的副本，那么 aObj == bObj 返回 true，a**.equal?**bObj 返回 false，但是 a**.equal?**aObj 返回 true。

Ruby 赋值运算符

假设变量 a 的值为 10，变量 b 的值为 20，那么：

运算符	描述	实例
=	简单的赋值运算符，把右操作数的值赋给左操作数	c = a + b 将把 a + b 的值赋给 c
+=	加且赋值运算符，把右操作数加上左操作数的结果赋值给左操作数	c += a 相当于 c = c + a
-=	减且赋值运算符，把左操作数减去右操作数的结果赋值给左操作数	c -= a 相当于 c = c - a
*=	乘且赋值运算符，把右操作数乘以左操作数的结果赋值给左操作数	c = *a* 相当于 *c* = *c* a
/=	除且赋值运算符，把左操作数除以右操作数的结果赋值给左操作数	c /= a 相当于 c = c / a
%=	求模且赋值运算符，求两个操作数的模赋值给左操作数	c %= a 相当于 c = c % a
**=	指数且赋值运算符，执行指数计算，并赋值给左操作数	c **= a 相当于 c = c ** a

Ruby 并行赋值

Ruby 也支持变量的并行赋值。这使得多个变量可以通过一行的 Ruby 代码进行初始化。例如：

```
a = 10
b = 20
c = 30
```

使用并行赋值可以更快地声明：

```
a, b, c = 10, 20, 30
```

并行赋值在交换两个变量的值时也很有用：

```
a, b = b, c
```

Ruby 位运算符

位运算符作用于位，并逐位执行操作。

假设如果 a = 60，且 b = 13，现在以二进制格式，它们如下所示：

```
a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011
```

下表列出了 Ruby 支持的位运算符。

运算符	描述	实例
&	如果同时存在于两个操作数中，二进制 AND 运算符复制一位到结果中。	(a & b) 将得到 12，即为 0000 1100
|	如果存在于任一操作数中，二进制 OR 运算符复制一位到结果中。	(a | b) 将得到 61，即为 0011 1101
^	如果存在于其中一个操作数中但不同时存在于两个操作数中，二进制异或运算符复制一位到结果中。	(a ^ b) 将得到 49，即为 0011 0001
~	二进制补码运算符是一元运算符，具有"翻转"位效果。	(~a) 将得到 -61，即为 1100 0011，2 的补码形式，带符号的二进制数。
<<	二进制左移运算符。左操作数的值向左移动右操作数指定的位数。	a << 2 将得到 240，即为 1111 0000
>>	二进制右移运算符。左操作数的值向右移动右操作数指定的位数。	a >> 2 将得到 15，即为 0000 1111

Ruby 逻辑运算符

下表列出了 Ruby 支持的逻辑运算符。

假设变量 a 的值为 10，变量 b 的值为 20，那么：

运算符	描述	实例
and	称为逻辑与运算符。如果两个操作数都为真，则条件为真。	(a and b) 为真。
or	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(a or b) 为真。
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。	(a && b) 为真。
||	称为逻辑或运算符。如果两个操作数中有任意一个非零，则条件为真。	(a || b) 为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	!(a && b) 为假。
not	称为逻辑非运算符。用来逆转操作数的逻辑状态。如果条件为真则逻辑非运算符将使其为假。	not(a && b) 为假。

Ruby 三元运算符

有一个以上的操作称为三元运算符。第一个计算表达式的真假值，然后根据这个结果决定执行后边两个语句中的一个。条件运算符的语法如下：

运算符	描述	实例
? :	条件表达式	如果条件为真 ? 则值为 X : 否则值为 Y

Ruby 范围运算符

在 Ruby 中，序列范围用于创建一系列连续的值 - 包含起始值、结束值（视情况而定）和它们之间的值。

在 Ruby 中，这些序列是使用 ".." 和 "..." 范围运算符来创建的。两点形式创建的范围包含起始值和结束值，三点形式创建的范围只包含起始值不包含结束值。

运算符	描述	实例
..	创建一个从开始点到结束点的范围（包含结束点）	1..10 创建从 1 到 10 的范围
...	创建一个从开始点到结束点的范围（不包含结束点）	1...10 创建从 1 到 9 的范围

Ruby defined? 运算符

defined? 是一个特殊的运算符，以方法调用的形式来判断传递的表达式是否已定义。它返回表达式的描述字符串，如果表达式未定义则返回 *nil*。

下面是 defined? 运算符的各种用法：

用法 1

```
defined? variable # 如果 variable 已经初始化，则为 True
```

例如：

```
foo = 42
defined? foo    # => "local-variable"
defined? $_     # => "global-variable"
defined? bar    # => nil（未定义）
```

用法 2

```
defined? method_call # 如果方法已经定义，则为 True
```

例如：

```
defined? puts        # => "method"
defined? puts(bar)   # => nil（在这里 bar 未定义）
defined? unpack      # => nil（在这里未定义）
```

用法 3

```
# 如果存在可被 super 用户调用的方法，则为 True
defined? super
```

例如：

```
defined? super     # => "super"（如果可被调用）
defined? super     # => nil（如果不可被调用）
```

用法 4

```
defined? yield   # 如果已传递代码块，则为 True
```

例如：

```
defined? yield    # => "yield"（如果已传递块）
defined? yield    # => nil（如果未传递块）
```

Ruby 点运算符 "." 和双冒号运算符 "::"

您可以通过在方法名称前加上模块名称和一条下划线来调用模块方法。您可以使用模块名称和两个冒号来引用一个常量。

:: 是一元运算符，允许在类或模块内定义常量、实例方法和类方法，可以从类或模块外的任何地方进行访问。

请记住：在 Ruby 中，类和方法也可以被当作常量。

您只需要在表达式的常量名前加上 :: 前缀，即可返回适当的类或模块对象。

如果未使用前缀表达式，则默认使用主 Object 类。

下面是两个实例：

```
MR_COUNT = 0        # 定义在主 Object 类上的常量
module Foo
  MR_COUNT = 0
  ::MR_COUNT = 1    # 设置全局计数为 1
  MR_COUNT = 2      # 设置局部计数为 2
end
puts MR_COUNT       # 这是全局常量
puts Foo::MR_COUNT  # 这是 "Foo" 的局部常量
```

第二个实例：

```
CONST = ' out there'
class Inside_one
   CONST = proc {' in there'}
   def where_is_my_CONST
      ::CONST + ' inside one'
   end
end
class Inside_two
   CONST = ' inside two'
   def where_is_my_CONST
      CONST
   end
end
puts Inside_one.new.where_is_my_CONST
puts Inside_two.new.where_is_my_CONST
puts Object::CONST + Inside_two::CONST
puts Inside_two::CONST + CONST
puts Inside_one::CONST
puts Inside_one::CONST.call + Inside_two::CONST
```

Ruby 运算符的优先级

下表按照运算符的优先级从高到低列出了所有的运算符。

方法	运算符	描述
是	::	常量解析运算符
是	[ ] [ ]=	元素引用、元素集合
是	**	指数
是	! ~ + -	非、补、一元加、一元减（最后两个的方法名为 +@ 和 -@）
是	* / %	乘法、除法、求模
是	+ -	加法和减法
是	>> <<	位右移、位左移
是	&	位与
是	^ |	位异或、位或
是	<= < > >=	比较运算符
是	<=> == === != =~ !~	相等和模式匹配运算符（!= 和 !~ 不能被定义为方法）
	&&	逻辑与
	||	逻辑或
	.. ...	范围（包含、不包含）
	? :	三元 if-then-else
	= %= { /= -= += |= &= >>= <<= *= &&= ||= **=	赋值
	defined?	检查指定符号是否已定义
	not	逻辑否定
	or and	逻辑组成

注意：在方法列标识为 是 的运算符实际上是方法，因此可以被重载。

Ruby 注释

注释是在运行时会被忽略的 Ruby 代码内的注释行。单行注释以 # 字符开始，直到该行结束，如下所示：

```
#!/usr/bin/ruby -w

# 这是一个单行注释。

puts "Hello, Ruby!"
```

当执行时，上面的程序会产生以下结果：

```
Hello, Ruby!
```

Ruby 多行注释

您可以使用 **=begin** 和 **=end** 语法注释多行，如下所示：

```
#!/usr/bin/ruby -w

puts "Hello, Ruby!"

=begin
这是一个多行注释。
可扩展至任意数量的行。
但 =begin 和 =end 只能出现在第一行和最后一行。
=end
```

当执行时，上面的程序会产生以下结果：

```
Hello, Ruby!
```

请确保尾部的注释离代码有足够的距离，以便容易区分注释和代码。如果块中超过一条尾部注释，请对齐它们。例如：

```
@counter      # 跟踪页面被击中的次数
@siteCounter  # 跟踪所有页面被击中的次数
```

Ruby 判断

Ruby 提供了其他现代语言中很常见的条件结构。在这里，我们将解释所有的条件语句和 Ruby 中可用的修饰符。

Ruby *if...else* 语句

语法

```
if conditional [then]
      code...
[elsif conditional [then]
      code...]...
[else
      code...]
end
```

if 表达式用于条件执行。值 *false* 和 *nil* 为假，其他值都为真。请注意，Ruby 使用 elsif，不是使用 else if 和 elif。

如果 *conditional* 为真，则执行 *code*。如果 *conditional* 不为真，则执行 else 子句中指定的 *code*。

if 表达式的 *conditional* 通过保留字 *then*、一个换行符或一个分号，来与代码分离开。

实例

```
#!/usr/bin/ruby

x=1
if x > 2
   puts "x is greater than 2"
elsif x <= 2 and x!=0
   puts "x is 1"
else
   puts "I can't guess the number"
end
```

```
x is 1
```

Ruby *if* 修饰符

语法

```
code if condition
```

如果 *conditional* 为真，则执行 *code*。

实例

```
#!/usr/bin/ruby

$debug=1
print "debug\n" if $debug
```

这将产生以下结果：

```
debug
```

Ruby *unless* 语句

语法

```
unless conditional [then]
   code
[else
   code ]
end
```

如果 *conditional* 为假，则执行 *code*。如果 *conditional* 为真，则执行 else 子句中指定的 *code*。

实例

```
#!/usr/bin/ruby

x=1
unless x>2
   puts "x is less than 2"
 else
  puts "x is greater than 2"
end
```

这将产生以下结果：

```
x is less than 2
```

Ruby *unless* 修饰符

语法

```
code unless conditional
```

如果 *conditional* 为假，则执行 *code*。

实例

```
#!/usr/bin/ruby

$var =  1
print "1 -- Value is set\n" if $var
print "2 -- Value is set\n" unless $var

$var = false
print "3 -- Value is set\n" unless $var
```

这将产生以下结果：

```
1 -- Value is set
3 -- Value is set
```

Ruby *case* 语句

语法

```
case expression
[when expression [, expression ...] [then]
   code ]...
[else
   code ]
end
```

比较 case 所指定的 *expression*，当使用 === 运算符指定时，执行匹配的 when 子句的 *code*。

when 子句所指定的 *expression* 背当作左操作数。如果没有匹配的 when 子句，*case* 执行 *else* 子句的代码。

when 语句的表达式通过保留字 *then*、一个换行符或一个分号，来与代码分离开。

因此：

```
case expr0
when expr1, expr2
   stmt1
when expr3, expr4
   stmt2
else
   stmt3
end
```

基本上类似于：

```
_tmp = expr0
if expr1 === _tmp || expr2 === _tmp
   stmt1
elsif expr3 === _tmp || expr4 === _tmp
   stmt2
else
   stmt3
end
```

实例

```
#!/usr/bin/ruby

$age =  5
case $age
when 0 .. 2
    puts "baby"
when 3 .. 6
    puts "little child"
when 7 .. 12
    puts "child"
when 13 .. 18
    puts "youth"
else
    puts "adult"
end
```

这将产生以下结果：

```
little child
```

Ruby 循环

Ruby 中的循环用于执行相同的代码块若干次。本章节将详细介绍 Ruby 支持的所有循环语句。

Ruby *while* 语句

语法

```
while conditional [do]
   code
end
```

当 *conditional* 为真时，执行 *code*。*while* 循环的 *conditional* 通过保留字 *do*、一个换行符、反斜线 \ 或一个分号 ; ，来与 *code* 分离开。

实例

```
#!/usr/bin/ruby

$i = 0
$num = 5

while $i < $num  do
   puts("Inside the loop i = #$i" )
   $i +=1
end
```

这将产生以下结果：

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby *while* 修饰符

语法

```
code while condition

或者

begin
  code
end while conditional
```

当 *conditional* 为真时，执行 *code*。

如果 *while* 修饰符跟在一个没有 *rescue* 或 ensure 子句的 *begin* 语句后面，*code* 会在 *conditional* 判断之前执行一次。

实例

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
   puts("Inside the loop i = #$i" )
   $i +=1
end while $i < $num
```

这将产生以下结果：

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
```

Ruby *until* 语句

```
until conditional [do]
   code
end
```

当 *conditional* 为假时，执行 *code*。*until* 语句的 *conditional* 通过保留字 *do*、一个换行符或一个分号，来与 *code* 分离开。

实例

```
#!/usr/bin/ruby

$i = 0
$num = 5

until $i > $num  do
   puts("Inside the loop i = #$i" )
   $i +=1;
end
```

这将产生以下结果：

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
Inside the loop i = 5
```

Ruby *until* 修饰符

语法

```
code until conditional

OR

begin
   code
end until conditional
```

当 *conditional* 为假时，执行 *code*。

如果 *until* 修饰符跟在一个没有 *rescue* 或 ensure 子句的 *begin* 语句后面，*code* 会在 *conditional* 判断之前执行一次。

实例

```
#!/usr/bin/ruby

$i = 0
$num = 5
begin
   puts("Inside the loop i = #$i" )
   $i +=1;
end until $i > $num
```

这将产生以下结果：

```
Inside the loop i = 0
Inside the loop i = 1
Inside the loop i = 2
Inside the loop i = 3
Inside the loop i = 4
Inside the loop i = 5
```

Ruby *for* 语句

语法

```
for variable [, variable ...] in expression [do]
   code
end
```

针对 *expression* 中的每个元素分别执行一次 *code*。

实例

```
#!/usr/bin/ruby

for i in 0..5
   puts "Value of local variable is #{i}"
end
```

在这里，我们已经定义了范围 0..5。语句 for i in 0..5 允许 i 的值从 0 到 5（包含 5）。这将产生以下结果：

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

for...in 循环几乎是完全等价于：

```
(expression).each do |variable[, variable...]| code end
```

但是，for 循环不会为局部变量创建一个新的作用域。for 循环的 *expression* 通过保留字 *do*、一个换行符或一个分号，来与 *code* 分离开。.

实例

```
#!/usr/bin/ruby

(0..5).each do |i|
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Ruby *break* 语句

语法

```
break
```

终止最内部的循环。如果在块内调用，则终止相关块的方法（方法返回 nil）。

实例

```
#!/usr/bin/ruby

for i in 0..5
   if i > 2 then
      break
   end
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 0
Value of local variable is 1
Value of local variable is 2
```

Ruby *next* 语句

语法

```
next
```

跳到最内部循环的下一个迭代。如果在块内调用，则终止块的执行（*yield* 或调用返回 nil）。

实例

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      next
   end
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果：

```
Value of local variable is 2
Value of local variable is 3
Value of local variable is 4
Value of local variable is 5
```

Ruby *redo* 语句

语法

```
redo
```

重新开始最内部循环的该次迭代，不检查循环条件。如果在块内调用，则重新开始 *yield* 或 *call*。

实例

```
#!/usr/bin/ruby

for i in 0..5
   if i < 2 then
      puts "Value of local variable is #{i}"
      redo
   end
end
```

这将产生以下结果，并会进入一个无限循环：

```
Value of local variable is 0
Value of local variable is 0
............................
```

Ruby *retry* 语句

语法

```
retry
```

如果 *retry* 出现在 begin 表达式的 rescue 子句中，则从 begin 主体的开头重新开始。

```
begin
   do_something # 抛出的异常
rescue
   # 处理错误
   retry  # 重新从 begin 开始
end
```

如果 retry 出现在迭代内、块内或者 for 表达式的主体内，则重新开始迭代调用。迭代的参数会重新评估。

```
for i in 1..5
   retry if some_condition # 重新从 i == 1 开始
end
```

实例

```
#!/usr/bin/ruby

for i in 1..5
   retry if  i > 2
   puts "Value of local variable is #{i}"
end
```

这将产生以下结果，并会进入一个无限循环：

```
Value of local variable is 1
Value of local variable is 2
Value of local variable is 1
Value of local variable is 2
Value of local variable is 1
Value of local variable is 2
............................
```

Ruby 方法

Ruby 方法与其他编程语言中的函数类似。Ruby 方法用于捆绑一个或多个重复的语句到一个单元中。

方法名应以小写字母开头。如果您以大写字母作为方法名的开头，Ruby 可能会把它当作常量，从而导致不正确地解析调用。

方法应在调用之前定义，否则 Ruby 会产生未定义的方法调用异常。

语法

```
def method_name [( [arg [= default]]...[, * arg [, &expr ]])]
   expr..
end
```

所以，您可以定义一个简单的方法，如下所示：

```
def method_name
   expr..
end
```

您可以定义一个接受参数的方法，如下所示：

```
def method_name (var1, var2)
   expr..
end
```

您可以为参数设置默认值，如果方法调用时未传递必需的参数则使用默认值：

```
def method_name (var1=value1, var2=value2)
   expr..
end
```

当您要调用方法时，只需要使用方法名即可，如下所示：

```
method_name
```

但是，当您调用带参数的方法时，您在写方法名时还要带上参数，例如：

```
method_name 25, 30
```

使用带参数方法最大的缺点是调用方法时需要记住参数个数。例如，如果您向一个接受三个参数的方法只传递了两个参数，Ruby 会显示错误。

实例

```
#!/usr/bin/ruby

def test(a1="Ruby", a2="Perl")
   puts "The programming language is #{a1}"
   puts "The programming language is #{a2}"
end
test "C", "C++"
test
```

这将产生以下结果：

```
The programming language is C
The programming language is C++
The programming language is Ruby
The programming language is Perl
```

从方法返回值

Ruby 中的每个方法默认都会返回一个值。这个返回的值是最后一个语句的值。例如：

```
def test
   i = 100
   j = 10
   k = 0
end
```

在调用这个方法时，将返回最后一个声明的变量 k。

Ruby *return* 语句

Ruby 中的 *return* 语句用于从 Ruby 方法中返回一个或多个值。

语法

```
return [expr[`,' expr...]]
```

如果给出超过两个的表达式，包含这些值的数组将是返回值。如果未给出表达式，nil 将是返回值。

实例

```
return

OR

return 12

OR

return 1,2,3
```

看看下面的实例：

```
#!/usr/bin/ruby

def test
   i = 100
   j = 200
   k = 300
return i, j, k
end
var = test
puts var
```

这将产生以下结果：

```
100
200
300
```

可变数量的参数

假设您声明了一个带有两个参数的方法，当您调用该方法时，您同时还需要传递两个参数。

但是，Ruby 允许您声明参数数量可变的方法。让我们看看下面的实例：

```
#!/usr/bin/ruby

def sample (*test)
   puts "The number of parameters is #{test.length}"
   for i in 0...test.length
      puts "The parameters are #{test[i]}"
   end
end
sample "Zara", "6", "F"
sample "Mac", "36", "M", "MCA"
```

在这段代码中，您已经声明了一个方法 sample，接受一个参数 test。但是，这个参数是一个变量参数。这意味着参数可以带有不同数量的变量。所以上面的代码将产生下面的结果：

```
The number of parameters is 3
The parameters are Zara
The parameters are 6
The parameters are F
The number of parameters is 4
The parameters are Mac
The parameters are 36
The parameters are M
The parameters are MCA
```

类方法

当方法定义在类定义外部时，方法默认标记为 *private*。另一方面，定义在类定义中的方法默认标记为 public。方法默认的可见性和 *private* 标记可通过模块（Module）的 *public* 或 *private* 改变。

当你想要访问类的方法时，您首先需要实例化类。然后，使用对象，您可以访问类的任何成员。

Ruby 提供了一种不用实例化类即可访问方法的方式。让我们看看如何声明并访问类方法：

```
class Accounts
   def reading_charge
   end
   def Accounts.return_date
   end
end
```

我们已经知道方法 return_date 是如何声明的。它是通过在类名后跟着一个点号，点号后跟着方法名来声明的。您可以直接访问类方法，如下所示：

```
Accounts.return_date
```

如需访问该方法，您不需要创建类 Accounts 的对象。

Ruby *alias* 语句

这个语句用于为方法或全局变量起别名。别名不能在方法主体内定义。即使方法被重写，方法的别名也保持方法的当前定义。

为编号的全局变量（$1, $2,...）起别名是被禁止的。重写内置的全局变量可能会导致严重的问题。

语法

```
alias method-name method-name
alias global-variable-name global-variable-name
```

实例

```
alias foo bar
alias $MATCH $&
```

在这里，我们已经为 bar 定义了别名为 foo，为 $& 定义了别名为 $MATCH。

Ruby *undef* 语句

这个语句用于取消方法定义。*undef* 不能出现在方法主体内。

通过使用 *undef* 和 *alias*，类的接口可以从父类独立修改，但请注意，在自身内部方法调用时，它可能会破坏程序。

语法

```
undef method-name
```

实例

下面的实例取消名为 *bar*的方法定义：

```
undef bar
```

Ruby 块

您已经知道 Ruby 如何定义方法以及您如何调用方法。类似地，Ruby 有一个块的概念。

- 块由大量的代码组成。
- 您需要给块取个名称。
- 块中的代码总是包含在大括号 {} 内。
- 块总是从与其具有相同名称的函数调用。这意味着如果您的块名称为 *test*，那么您要使用函数 *test* 来调用这个块。
- 您可以使用 *yield* 语句来调用块。

语法

```
block_name{
   statement1
   statement2
   ..........
}
```

在这里，您将学到如何使用一个简单的 *yield* 语句来调用块。您也将学到如何使用带有参数的 *yield* 语句来调用块。在实例中，您将看到这两种类型的 *yield* 语句。

yield 语句

让我们看一个 yield 语句的实例：

```
#!/usr/bin/ruby

def test
   puts "You are in the method"
   yield
   puts "You are again back to the method"
   yield
end
test {puts "You are in the block"}
```

这将产生以下结果：

```
You are in the method
You are in the block
You are again back to the method
You are in the block
```

您也可以传递带有参数的 yield 语句。下面是一个实例：

```
#!/usr/bin/ruby

def test
   yield 5
   puts "You are in the method test"
   yield 100
end
test {|i| puts "You are in the block #{i}"}
```

这将产生以下结果：

```
You are in the block 5
You are in the method test
You are in the block 100
```

在这里，*yield* 语句后跟着参数。您甚至可以传递多个参数。在块中，您可以在两个竖线之间放置一个变量来接受参数。因此，在上面的代码中，yield 5 语句向 test 块传递值 5 作为参数。

现在，看下面的语句：

```
test {|i| puts "You are in the block #{i}"}
```

在这里，值 5 会在变量 i 中收到。现在，观察下面的 puts 语句：

```
puts "You are in the block #{i}"
```

这个 puts 语句的输出是：

```
You are in the block 5
```

如果您想要传递多个参数，那么 *yield* 语句如下所示：

```
yield a, b
```

此时，块如下所示：

```
test {|a, b| statement}
```

参数使用逗号分隔。

块和方法

您已经看到块和方法之间是如何相互关联的。您通常使用 yield 语句从与其具有相同名称的方法调用块。因此，代码如下所示：

```
#!/usr/bin/ruby

def test
  yield
end
test{ puts "Hello world"}
```

本实例是实现块的最简单的方式。您使用 *yield* 语句调用 test 块。

但是如果方法的最后一个参数前带有 &，那么您可以向该方法传递一个块，且这个块可被赋给最后一个参数。如果 * 和 & 同时出现在参数列表中，& 应放在后面。

```
#!/usr/bin/ruby

def test(&block)
   block.call
end
test { puts "Hello World!"}
```

这将产生以下结果：

```
Hello World!
```

BEGIN 和 END 块

每个 Ruby 源文件可以声明当文件被加载时要运行的代码块（BEGIN 块），以及程序完成执行后要运行的代码块（END 块）。

```
#!/usr/bin/ruby

BEGIN {
  # BEGIN block code
  puts "BEGIN code block"
}

END {
  # END block code
  puts "END code block"
}
  # MAIN block code
puts "MAIN code block"
```

一个程序可以包含多个 BEGIN 和 END 块。BEGIN 块按照它们出现的顺序执行。END 块按照它们出现的相反顺序执行。当执行时，上面的程序产生产生以下结果：

```
BEGIN code block
MAIN code block
END code block
```

Ruby 模块（Module）

模块（Module）是一种把方法、类和常量组合在一起的方式。模块（Module）为您提供了两大好处。

- 模块提供了一个命名空间和避免名字冲突。
- 模块实现了 *mixin* 装置。

模块（Module）定义了一个命名空间，相当于一个沙箱，在里边您的方法和常量不会与其他地方的方法常量冲突。

语法

```
module Identifier
   statement1
   statement2
   ...........
end
```

模块常量命名与类常量命名类似，以大写字母开头。方法定义看起来也相似：模块方法定义与类方法定义类似。

通过类方法，您可以在类方法名称前面放置模块名称和一个点号来调用模块方法，您可以使用模块名称和两个冒号来引用一个常量。

实例

```
#!/usr/bin/ruby

# 定义在 trig.rb 文件中的模块

module Trig
   PI = 3.141592654
   def Trig.sin(x)
   # ..
   end
   def Trig.cos(x)
   # ..
   end
end
```

我们可以定义多个函数名称相同但是功能不同的模块：

```
#!/usr/bin/ruby

# 定义在 moral.rb 文件中的模块

module Moral
   VERY_BAD = 0
   BAD = 1
   def Moral.sin(badness)
   # ...
   end
end
```

就像类方法，当您在模块中定义一个方法时，您可以指定在模块名称后跟着一个点号，点号后跟着方法名。

Ruby *require* 语句

require 语句类似于 C 和 C++ 中的 include 语句以及 Java 中的 import 语句。如果一个第三方的程序想要使用任何已定义的模块，则可以简单地使用 Ruby *require* 语句来加载模块文件：

语法

```
require filename
```

在这里，文件扩展名 **.rb** 不是必需的。

实例

```
$LOAD_PATH << '.'

require 'trig.rb'
require 'moral'

y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

在这里，我们使用 **$LOAD_PATH << '.'** 让 Ruby 知道必须在当前目录中搜索被引用的文件。如果您不想使用 $LOAD_PATH，那么您可以使用 **require_relative** 来从一个相对目录引用文件。

注意：在这里，文件包含相同的函数名称。所以，这会在引用调用程序时导致代码模糊，但是模块避免了这种代码模糊，而且我们可以使用模块的名称调用适当的函数。

Ruby *include* 语句

您可以在类中嵌入模块。为了在类中嵌入模块，您可以在类中使用 *include* 语句：

语法

```
include modulename
```

如果模块是定义在一个单独的文件中，那么在嵌入模块之前使用 *require* 语句引用该文件时必需的。

实例

假设下面的模块写在 *support.rb* 文件中。

```
module Week
   FIRST_DAY = "Sunday"
   def Week.weeks_in_month
      puts "You have four weeks in a month"
   end
   def Week.weeks_in_year
      puts "You have 52 weeks in a year"
   end
end
```

现在，您可以在类中引用该模块，如下所示：

```
#!/usr/bin/ruby
$LOAD_PATH << '.'
require "support"

class Decade
include Week
   no_of_yrs=10
   def no_of_months
      puts Week::FIRST_DAY
      number=10*12
      puts number
   end
end
d1=Decade.new
puts Week::FIRST_DAY
Week.weeks_in_month
Week.weeks_in_year
d1.no_of_months
```

这将产生以下结果：

```
Sunday
You have four weeks in a month
You have 52 weeks in a year
Sunday
120
```

Ruby 中的 Mixins

在阅读本节之前，您需要初步了解面向对象的概念。

当一个类可以从多个父类继承类的特性时，该类显示为多重继承。

Ruby 不直接支持多重继承，但是 Ruby 的模块（Module）有另一个神奇的功能。它几乎消除了多重继承的需要，提供了一种名为 *mixin* 的装置。

Mixins 向您提供了一种完美的为类添加功能的控制方式。但是，它们真正的强大在于当 mixin 中的代码开始与使用它的类中的代码交互时。

让我们看看下面的示例代码，深入了解 mixin：

```
module A
   def a1
   end
   def a2
   end
end
module B
   def b1
   end
   def b2
   end
end

class Sample
include A
include B
   def s1
   end
end

samp=Sample.new
samp.a1
samp.a2
samp.b1
samp.b2
samp.s1
```

模块 A 由方法 a1 和 a2 组成。模块 B 由方法 b1 和 b2 组成。类 Sample 包含了模块 A 和 B。类 Sample 可以访问所有四个方法，即 a1、a2、b1 和 b2。因此，您可以看到类 Sample 继承了两个模块。因此，您可以说类 Sample 显示了多重继承或 *mixin* 。

Ruby 字符串（String）

Ruby 中的 String 对象存储并操作一个或多个字节的任意序列，通常表示那些代表人类语言的字符。

最简单的字符串是括在单引号（单引号字符）内。在引号标记内的文本是字符串的值：

```
'This is a simple Ruby string literal'
```

如果您需要在单引号字符串内使用单引号字符，那么需要在单引号字符串使用反斜杠，这样 Ruby 解释器就不会认为这个单引号字符会终止字符串：

```
'Won\'t you read O\'Reilly\'s book?'
```

反斜杠也能转义另一个反斜杠，这样第二个反斜杠本身不会解释为转义字符。

以下是 Ruby 中字符串相关的特性。

表达式置换

表达式置换是一种使用 #{ 和 } 把任意 Ruby 表达式的值嵌入到字符串中的方式：

```
#!/usr/bin/ruby

x, y, z = 12, 36, 72
puts "The value of x is #{ x }."
puts "The sum of x and y is #{ x + y }."
puts "The average was #{ (x + y + z)/3 }."
```

这将产生以下结果：

```
The value of x is 12.
The sum of x and y is 48.
The average was 40.
```

一般的分隔字符串

通过一般的分隔字符串，您可以在以百分号字符（%）为前导的一对匹配的任意分隔字符（例如，!、 (、 {、 <，等等）内创建字符串。Q、 q 和 x 有特殊的意义。一般的分隔字符串可以是：

```
%{Ruby is fun.}  相当于 "Ruby is fun."
%Q{ Ruby is fun. } 相当于 " Ruby is fun. "
%q[Ruby is fun.]  相当于以单引号字符串
%x!ls! 相当于反勾号命令输出 `ls`
```

转义字符

下标列出了可使用反斜杠符号转义的转义字符或非打印字符。

注意：在一个双引号括起的字符串内，转义字符会被解释；在一个单引号括起的字符串内，转义字符会被保留。

反斜杠符号	十六进制字符	描述
\a	0x07	报警符
\b	0x08	退格键
\cx		Control-x
\C-x		Control-x
\e	0x1b	转义符
\f	0x0c	换页符
\M-\C-x		Meta-Control-x
\n	0x0a	换行符
\nnn		八进制表示法，其中 n 的范围为 0.7
\r	0x0d	回车符
\s	0x20	空格符
\t	0x09	制表符
\v	0x0b	垂直制表符
\x		字符 x
\xnn		十六进制表示法，其中 n 的范围为 0.9、 a.f 或 A.F

字符编码

Ruby 的默认字符集是 ASCII，字符可用单个字节表示。如果您使用 UTF-8 或其他现代的字符集，字符可能是用一个到四个字节表示。

您可以在程序开头使用 $KCODE 改变字符集，如下所示：

```
$KCODE = 'u'
```

下面是 $KCODE 可能的值。

编码	描述
a	ASCII （与 none 相同）。这是默认的。
e	EUC。
n	None （与 ASCII 相同）。
u	UTF-8。

字符串内建方法

我们需要有一个 String 对象的实例来调用 String 方法。下面是创建 String 对象实例的方式：

```
new [String.new(str="")]
```

这将返回一个包含 *str* 副本的新的字符串对象。现在，使用 *str* 对象，我们可以调用任意可用的实例方法。例如：

```
#!/usr/bin/ruby

myStr = String.new("THIS IS TEST")
foo = myStr.downcase

puts "#{foo}"
```

这将产生以下结果：

```
this is test
```

下面是公共的字符串方法（假设 str 是一个 String 对象）：

方法	描述
str % arg	使用格式规范格式化字符串。如果 arg 包含一个以上的替代，那么 arg 必须是一个数组。如需了解更多格式规范的信息，请查看"内核模块"下的 sprintf。
str * integer	返回一个包含 integer 个 str 的新的字符串。换句话说，str 被重复了 integer 次。
str + other_str	连接 other_str 到 str。
str << obj	连接一个对象到字符串。如果对象是范围为 0.255 之间的固定数字 Fixnum，则它会被转换为一个字符。把它与 concat 进行比较。
str <=> other_str	把 str 与 other_str 进行比较，返回 -1（小于）、0（等于）或 1（大于）。比较是区分大小写的。

str == obj	检查 str 和 obj 的相等性。如果 obj 不是字符串，则返回 false，如果 str <=> obj，则返回 true，返回 0。
str =~ obj	根据正则表达式模式 obj 匹配 str。返回匹配开始的位置，否则返回 false。
str =~ obj	根据正则表达式模式 obj 匹配 str。返回匹配开始的位置，否则返回 false。
str.capitalize	把字符串转换为大写字母显示。
str.capitalize!	与 capitalize 相同，但是 str 会发生变化并返回。
str.casecmp	不区分大小写的字符串比较。
str.center	居中字符串。
str.chomp	从字符串末尾移除记录分隔符（$/），通常是 \n。如果没有记录分隔符，则不进行任何操作。
str.chomp!	与 chomp 相同，但是 str 会发生变化并返回。
str.chop	移除 str 中的最后一个字符。
str.chop!	与 chop 相同，但是 str 会发生变化并返回。
str.concat(other_str)	连接 other_str 到 str。
str.count(str, ...)	给一个或多个字符集计数。如果有多个字符集，则给这些集合的交集计数。
str.crypt(other_str)	对 str 应用单向加密哈希。参数是两个字符长的字符串，每个字符的范围为 a.z、 A.Z、 0.9、 . 或 /。
str.delete(other_str, ...)	返回 str 的副本，参数交集中的所有字符会被删除。
str.delete!(other_str, ...)	与 delete 相同，但是 str 会发生变化并返回。
str.downcase	返回 str 的副本，所有的大写字母会被替换为小写字母。
str.downcase!	与 downcase 相同，但是 str 会发生变化并返回。
str.dump	返回 str 的版本，所有的非打印字符被替换为 \nnn 符号，所有的特殊字符被转义。
str.each(separator=$/) { |substr| block }	使用参数作为记录分隔符（默认是 $/）分隔 str，传递每个子字符串给被提供的块。
str.each_byte { |fixnum| block }	传递 str 的每个字节给 block，以字节的十进制表示法返回每个字节。
str.each_line(separator=$/) { |substr| block }	使用参数作为记录分隔符（默认是 $/）分隔 str，传递每个子字符串给被提供的 block。
str.empty?	如果 str 为空（即长度为 0），则返回 true。
str.eql?(other)	如果两个字符串有先攻的长度和内容，则这两个字符串相等。
	返回 str 的副本，pattern 的所有出现都替换为

replacement) [or] str.gsub(pattern) { |match| block }	replacement 或 block 的值。pattern 通常是一个正则表达式 Regexp；如果是一个字符串 String，则没有正则表达式元字符被解释（即，/\d/ 将匹配一个数字，但 '\d' 将匹配一个反斜杠后跟一个 'd'）。
str[fixnum] [or] str[fixnum,fixnum] [or] str[range] [or] str[regexp] [or] str[regexp, fixnum] [or] str[other_str]	使用下列的参数引用 str：参数为一个 Fixnum，则返回 fixnum 的字符编码；参数为两个 Fixnum，则返回一个从偏移（第一个 fixnum）开始截至到长度（第二个 fixnum）为止的子字符串；参数为 range，则返回该范围内的一个子字符串；参数为 regexp，则返回匹配字符串的部分；参数为带有 fixnum 的 regexp，则返回 fixnum 位置的匹配数据；参数为 other_str，则返回匹配 other_str 的子字符串。一个负数的 Fixnum 从字符串的末尾 -1 开始。
str[fixnum] = fixnum [or] str[fixnum] = new_str [or] str[fixnum, fixnum] = new_str [or] str[range] = aString [or] str[regexp] =new_str [or] str[regexp, fixnum] =new_str [or] str[other_str] = new_str]	替换整个字符串或部分字符串。与 slice! 同义。
str.gsub!(pattern, replacement) [or] str.gsub!(pattern) { |match| block }	执行 String#gsub 的替换，返回 str，如果没有替换被执行则返回 nil。
str.hash	返回一个基于字符串长度和内容的哈希。
str.hex	把 str 的前导字符当作十六进制数字的字符串（一个可选的符号和一个可选的 0x），并返回相对应的数字。如果错误则返回零。
str.include? other_str [or] str.include? fixnum	如果 str 包含给定的字符串或字符，则返回 true。
str.index(substring [, offset]) [or] str.index(fixnum [, offset]) [or] str.index(regexp [, offset])	返回给定子字符串、字符（fixnum）或模式（regexp）在 str 中第一次出现的索引。如果未找到则返回 nil。如果提供了第二个参数，则指定在字符串中开始搜索的位置。
str.insert(index, other_str)	在给定索引的字符前插入 other_str，修改 str。负值索引从字符串的末尾开始计数，并在给定字符后插入。其意图是在给定的索引处开始插入一个字符串。
str.inspect	返回 str 的可打印版本，带有转义的特殊字符。
str.intern [or] str.to_sym	返回与 str 相对应的符号，如果之前不存在，则创建符号。
str.length	返回 str 的长度。把它与 size 进行比较。
str.ljust(integer, padstr=' ')	如果 integer 大于 str 的长度，则返回长度为 integer 的新字符串，新字符串以 str 左对齐，并以 padstr 作为填充。否则，返回 str。

	充。否则，返回 str。
str.lstrip	返回 str 的副本，移除了前导的空格。
str.lstrip!	从 str 中移除前导的空格，如果没有变化则返回 nil。
str.match(pattern)	如果 pattern 不是正则表达是，则把 pattern 转换为正则表达式 Regexp，然后在 str 上调用它的匹配方法。
str.oct	把 str 的前导字符当作十进制数字的字符串（一个可选的符号），并返回相对应的数字。如果转换失败，则返回 0。
str.replace(other_str)	把 str 中的内容替换为 other_str 中的相对应的值。
str.reverse	返回一个新字符串，新字符串是 str 的倒序。
str.reverse!	逆转 str，str 会发生变化并返回。
str.rindex(substring [, fixnum]) [or] str.rindex(fixnum [, fixnum]) [or] str.rindex(regexp [, fixnum])	返回给定子字符串、字符（fixnum）或模式（regexp）在 str 中最后一次出现的索引。如果未找到则返回 nil。如果提供了第二个参数，则指定在字符串中结束搜索的位置。超出该点的字符将不被考虑。
str.rjust(integer, padstr=' ')	如果 integer 大于 str 的长度，则返回长度为 integer 的新字符串，新字符串以 str 右对齐，并以 padstr 作为填充。否则，返回 str。
str.rstrip	返回 str 的副本，移除了尾随的空格。
str.rstrip!	从 str 中移除尾随的空格，如果没有变化则返回 nil。
str.scan(pattern) [or] str.scan(pattern) { |match, ...| block }	两种形式匹配 pattern（可以是一个正则表达式 Regexp 或一个字符串 String）遍历 str。针对每个匹配，会生成一个结果，结果会添加到结果数组中或传递给 block。如果 pattern 不包含分组，则每个独立的结果由匹配的字符串、>< 组成。如果 pattern 包含分组，每个独立的结果是一个包含每个分组入口的数组。
str.slice(fixnum) [or] str.slice(fixnum, fixnum) [or] str.slice(range) [or] str.slice(regexp) [or] str.slice(regexp, fixnum) [or] str.slice(other_str) See str[fixnum], etc. str.slice!(fixnum) [or] str.slice!(fixnum, fixnum) [or] str.slice!(range) [or] str.slice!(regexp) [or] str.slice!(other_str)	从 str 中删除指定的部分，并返回删除的部分。如果值超出范围，参数带有 Fixnum 的形式，将生成一个 IndexError。参数为 range 的形式，将生成一个 RangeError，参数为 Regexp 和 String 的形式，将忽略执行动作。
	基于分隔符，把 str 分成子字符串，并返回这些子字符串的数组。 如果 *pattern* 是一个字符串 String，那么在分割 str 时，它将作为分隔符使用。如果 pattern 是一个单

str.split(pattern=$;, [limit])	格和连续空格字符。 如果 *pattern* 是一个正则表达式 Regexp，则 str 在 pattern 匹配的地方被分割。当 pattern 匹配一个玲长度的字符串时，str 被分割成单个字符。 如果省略了 *pattern* 参数，则使用 $; 的值。如果 $; 为 nil（默认的），str 基于空格进行分割，就像是指定了　　作为分隔符一样。 如果省略了 *limit* 参数，会抑制尾随的 null 字段。如果 limit 是一个正数，则最多返回该数量的字段（如果 limit 为 1，则返回整个字符串作为数组中的唯一入口）。如果 limit 是一个负数，则返回的字段数量不限制，且不抑制尾随的 null 字段。
str.squeeze([other_str]*)	使用为 String#count 描述的程序从 other_str 参数建立一系列字符。返回一个新的字符串，其中集合中出现的相同的字符会被替换为单个字符。如果没有给出参数，则所有相同的字符都被替换为单个字符。
str.squeeze!([other_str]*)	与 squeeze 相同，但是 str 会发生变化并返回，如果没有变化则返回 nil。
str.strip	返回 str 的副本，移除了前导的空格和尾随的空格。
str.strip!	从 str 中移除前导的空格和尾随的空格，如果没有变化则返回 nil。
str.sub(pattern, replacement) [or] str.sub(pattern) { |match| block }	返回 str 的副本，pattern 的第一次出现会被替换为 replacement 或 block 的值。pattern 通常是一个正则表达式 Regexp；如果是一个字符串 String，则没有正则表达式元字符被解释。
str.sub!(pattern, replacement) [or] str.sub!(pattern) { |match| block }	执行 String#sub 替换，并返回 str，如果没有替换执行，则返回 nil。
str.succ [or] str.next	返回 str 的继承。
str.succ! [or] str.next!	相当于 String#succ，但是 str 会发生变化并返回。
str.sum(n=16)	返回 str 中字符的 n-bit 校验和，其中 n 是可选的 Fixnum 参数，默认为 16。结果是简单地把 str 中每个字符的二进制值的总和，以 2n - 1 为模。这不是一个特别好的校验和。
str.swapcase	返回 str 的副本，所有的大写字母转换为小写字母，所有的小写字母转换为大写字母。
str.swapcase!	相当于 String#swapcase，但是 str 会发生变化并返回，如果没有变化则返回 nil。
str.to_f	返回把 str 中的前导字符解释为浮点数的结果。超出有效数字的末尾的多余字符会被忽略。如果在 str 的开头没有有效数字，则返回 0.0。该方法不会生成异常。
str.to_i(base=10)	返回把 str 中的前导字符解释为整数基数（基数为 2、8、 10 或 16）的结果。超出有效数字的末尾的多余字符会被忽略。如果在 str 的开头没有有效数字，则返回 0。该方法不会生成异常。

str.to_s [or] str.to_str	返回接收的值。
str.tr(from_str, to_str)	返回 str 的副本，把 from_str 中的字符替换为 to_str 中相对应的字符。如果 to_str 比 from_str 短，那么它会以最后一个字符进行填充。两个字符串都可以使用 c1.c2 符号表示字符的范围。如果 from_str 以 ^ 开头，则表示除了所列出的字符以外的所有字符。
str.tr!(from_str, to_str)	相当于 String#tr，但是 str 会发生变化并返回，如果没有变化则返回 nil。
str.tr_s(from_str, to_str)	把 str 按照 String#tr 描述的规则进行处理，然后移除会影响翻译的重复字符。
str.tr_s!(from_str, to_str)	相当于 String#tr_s，但是 str 会发生变化并返回，如果没有变化则返回 nil。
str.unpack(format)	根据 format 字符串解码 str（可能包含二进制数据），返回被提取的每个值的数组。format 字符由一系列单字符指令组成。每个指令后可以跟着一个数字，表示重复该指令的次数。星号（*）将使用所有剩余的元素。指令 sSiIlL 每个后可能都跟着一个下划线（_），为指定类型使用底层平台的本地尺寸大小，否则使用独立于平台的一致的尺寸大小。format 字符串中的空格会被忽略。
str.upcase	返回 str 的副本，所有的小写字母会被替换为大写字母。操作是环境不敏感的，只有字符 a 到 z 会受影响。
str.upcase!	改变 str 的内容为大写，如果没有变化则返回 nil。
str.upto(other_str) { |s| block }	遍历连续值，以 str 开始，以 other_str 结束（包含），轮流传递每个值给 block。String#succ 方法用于生成每个值。

字符串 unpack 指令 下表列出了方法 String#unpack 的解压指令。

指令	返回	描述
A	String	移除尾随的 null 和空格。
a	String	字符串。
B	String	从每个字符中提取位（首先是最高有效位）。
b	String	从每个字符中提取位（首先是最低有效位）。
C	Fixnum	提取一个字符作为无符号整数。
c	Fixnum	提取一个字符作为整数。
D, d	Float	把 sizeof(double) 长度的字符当作原生的 double。

E	Float	把 sizeof(double) 长度的字符当作 littleendian 字节顺序的 double。
e	Float	把 sizeof(float) 长度的字符当作 littleendian 字节顺序的 float。
F, f	Float	把 sizeof(float) 长度的字符当作原生的 float。
G	Float	把 sizeof(double) 长度的字符当作 network 字节顺序的 double。
g	Float	把 sizeof(float) 长度的字符当作 network 字节顺序的 float。
H	String	从每个字符中提取十六进制（首先是最高有效位）。
h	String	从每个字符中提取十六进制（首先是最低有效位）。
I	Integer	把 sizeof(int) 长度（通过 _ 修改）的连续字符当作原生的 integer。
i	Integer	把 sizeof(int) 长度（通过 _ 修改）的连续字符当作有符号的原生的 integer。
L	Integer	把四个（通过 _ 修改）连续字符当作无符号的原生的 long integer。
l	Integer	把四个（通过 _ 修改）连续字符当作有符号的原生的 long integer。
M	String	引用可打印的。
m	String	Base64 编码。
N	Integer	把四个字符当作 network 字节顺序的无符号的 long。
n	Fixnum	把两个字符当作 network 字节顺序的无符号的 short。
P	String	把 sizeof(char *) 长度的字符当作指针，并从引用的位置返回 \emph{len} 字符。
p	String	把 sizeof(char *) 长度的字符当作一个空结束字符的指针。
Q	Integer	把八个字符当作无符号的 quad word（64 位）。
q	Integer	把八个字符当作有符号的 quad word（64 位）。
S	Fixnum	把两个（如果使用 _ 则不同）连续字符当作 native 字节顺序的无符号的 short。
s	Fixnum	把两个（如果使用 _ 则不同）连续字符当作 native 字节顺序的有符号的 short。
U	Integer	UTF-8 字符，作为无符号整数。
u	String	UU 编码。
V	Fixnum	把四个字符当作 little-endian 字节顺序的无符号的 long。
v	Fixnum	把两个字符当作 little-endian 字节顺序的无符号的 short。
w	Integer	BER 压缩的整数。
X		向后跳过一个字符。
x		向前跳过一个字符。
Z	String	和 * 一起使用，移除尾随的 null 直到第一个 null。

@		跳过 length 参数给定的偏移量。

实例

尝试下面的实例，解压各种数据。

```
"abc \0\0abc \0\0".unpack('A6Z6')   #=> ["abc", "abc "]
"abc \0\0".unpack('a3a3')           #=> ["abc", " \000\000"]
"abc \0abc \0".unpack('Z*Z*')       #=> ["abc ", "abc "]
"aa".unpack('b8B8')                 #=> ["10000110", "01100001"]
"aaa".unpack('h2H2c')               #=> ["16", "61", 97]
"\xfe\xff\xfe\xff".unpack('sS')     #=> [-2, 65534]
"now=20is".unpack('M*')             #=> ["now is"]
"whole".unpack('xax2aX2aX1aX2a')    #=> ["h", "e", "l", "l", "o"]
```

Ruby 数组（Array）

Ruby 数组是任何对象的有序的、整数索引的集合。数组中的每个元素都与一个索引相关，并可通过索引进行获取。

数组的索引从 0 开始，这与 C 或 Java 中一样。一个负数的索引时相对于数组的末尾计数的，也就是说，索引为 -1 表示数组的最后一个元素，-2 表示数组中的倒数第二个元素，依此类推。

Ruby 数组可存储诸如 String、 Integer、 Fixnum、 Hash、 Symbol 等对象，甚至可以是其他 Array 对象。Ruby 数组不像其他语言中的数组那么刚性。当向数组添加元素时，Ruby 数组会自动增长。

创建数组

有多种方式创建或初始化数组。一种方式是通过 *new* 类方法：

```
names = Array.new
```

您可以在创建数组的同时设置数组的大小：

```
names = Array.new(20)
```

数组 *names* 的大小或长度为 20 个元素。您可以使用 size 或 length 方法返回数组的大小：

```
#!/usr/bin/ruby

names = Array.new(20)
puts names.size  # 返回 20
puts names.length # 返回 20
```

这将产生以下结果：

```
20
20
```

您可以给数组中的每个元素赋值，如下所示：

```
#!/usr/bin/ruby

names = Array.new(4, "mac")

puts "#{names}"
```

这将产生以下结果：

```
macmacmacmac
```

您也可以使用带有 new 的块，每个元素使用块中的计算结果来填充：

```
#!/usr/bin/ruby

nums = Array.new(10) { |e| e = e * 2 }

puts "#{nums}"
```

这将产生以下结果：

```
024681012141618
```

数组还有另一种方法，[]，如下所示：

```
nums = Array.[](1, 2, 3, 4,5)
```

数组创建的另一种形式如下所示：

```
nums = Array[1, 2, 3, 4,5]
```

在核心 Ruby 中可用的 *Kernel* 模块有一个 Array 方法，只接受单个参数。在这里，该方法带有一个范围作为参数来创建一个数字数组：

```
#!/usr/bin/ruby

digits = Array(0..9)

puts "#{digits}"
```

这将产生以下结果：

```
0123456789
```

数组内建方法

我们需要有一个 Array 对象的实例来调用 Array 方法。下面是创建 Array 对象实例的方式：

```
Array.[](...) [or] Array[...] [or] [...]
```

这将返回一个使用给定对象进行填充的新的数组。现在，使用创建的对象，我们可以调用任意可用的实例方法。例如：

```
#!/usr/bin/ruby

digits = Array(0..9)

num = digits.at(6)

puts "#{num}"
```

这将产生以下结果：

```
6
```

下面是公共的数组方法（假设 *array* 是一个 Array 对象）：

方法	描述
array & other_array	返回一个新的数组，包含两个数组中共同的元素，没有重复。
array ***int [or] array*** **str**	返回一个新的数组，新数组通过连接 self 的 int 副本创建的。带有 String 参数时，相当于 self.join(str)。
array + other_array	返回一个新的数组，新数组通过连接两个数组产生第三个数组创建的。
array - other_array	返回一个新的数组，新数组是从初始数组中移除了在 other_array 中出现的项的副本。
str <=> other_str	把 str 与 other_str 进行比较，返回 -1（小于）、0（等于）或 1（大于）。比较是区分大小写的。
array | other_array	通过把 other_array 加入 array 中，移除重复项，返回一个新的数组。
array << obj	把给定的对象附加到数组的末尾。该表达式返回数组本身，所以几个附加可以连在一起。
array <=> other_array	如果数组小于、等于或大于 other_array，则返回一个整数（-1、 0 或 +1）。
array == other_array	如果两个数组包含相同的元素个数，且每个元素与另一个数组中相对应的元素相等（根据 Object.==），那么这两个数组相等。
array[index] [or] array[start, length] [or] array[range] [or] array.slice(index) [or] array.slice(start, length) [or] array.slice(range)	返回索引为 *index* 的元素，或者返回从 *start* 开始直至 *length* 个元素的子数组，或者返回 *range* 指定的子数组。负值索引从数组末尾开始计数（-1 是最后一个元素）。如果 *index*（或开始索引）超出范围，则返回 *nil*。
array[index] = obj [or] array[start, length] = obj or	设置索引为 *index* 的元素，或者替换从 *start* 开始直至 *length* 个元素的子数组，或者替换 *range* 指定的子数

array[start, length] = obj or an_array or nil [or] array[range] = obj or an_array or nil	组。如果索引大于数组的当前容量，那么数组会自动增长。负值索引从数组末尾开始计数。如果 *length* 为零则插入元素。如果在第二种或第三种形式中使用了 *nil*，则从 *self* 删除元素。
array.abbrev(pattern = nil)	为 *self* 中的字符串计算明确的缩写集合。如果传递一个模式或一个字符串，只考虑当字符串匹配模式或者以该字符串开始时的情况。
array.assoc(obj)	搜索一个数组，其元素也是数组，使用 obj.== 把 obj 与每个包含的数组的第一个元素进行比较。如果匹配则返回第一个包含的数组，如果未找到匹配则返回 *nil*。
array.at(index)	返回索引为 index 的元素。一个负值索引从 *self* 的末尾开始计数。如果索引超出范围则返回 nil。
array.clear 从数组中移除所有的元素。	
array.collect { |item| block } [or] array.map { |item| block }	为 *self* 中的每个元素调用一次 *block*。创建一个新的数组，包含 block 返回的值。
array.collect! { |item| block } [or] array.map! { |item| block }	为 *self* 中的每个元素调用一次 *block*，把元素替换为 *block* 返回的值。
array.compact	返回 *self* 的副本，移除了所有的 *nil* 元素。
array.compact!	从数组中移除所有的 *nil* 元素。如果没有变化则返回 *nil*。
array.concat(other_array)	追加 other*array* 中的元素到 _*self* 中。
array.delete(obj) [or] array.delete(obj) { block }	从 *self* 中删除等于 *obj* 的项。如果未找到相等项，则返回 *nil*。如果未找到相等项且给出了可选的代码 *block*，则返回 *block* 的结果。
array.delete_at(index)	删除指定的 *index* 处的元素，并返回该元素。如果 index 超出范围，则返回 *nil*。
array.delete_if { |item| block }	当 *block* 为 true 时，删除 *self* 的每个元素。
array.each { |item| block }	为 *self* 中的每个元素调用一次 *block*，传递该元素作为参数。
array.each_index { |index| block }	与 Array#each 相同，但是传递元素的 *index*，而不是传递元素本身。
array.empty?	如果数组本身没有包含元素，则返回 true。
array.eql?(other)	如果 *array* 和 *other* 是相同的对象，或者两个数组带有相同的内容，则返回 true。
array.fetch(index) [or] array.fetch(index, default) [or] array.fetch(index) {	尝试返回位置 *index* 处的元素。如果 *index* 位于数组外部，则第一种形式会抛出 *IndexError* 异常，第二种形式会返回 *default*，第三种形式会返回调用 *block* 传入

|index| block }	*index* 的值。负值的 *index* 从数组末尾开始计数。
array.fill(obj) [or] array.fill(obj, start [, length]) [or] array.fill(obj, range) [or] array.fill { |index| block } [or] array.fill(start [, length]) { |index| block } [or] array.fill(range) { |index| block }	前面三种形式设置 *self* 的被选元素为 *obj*。以 *nil* 开头相当于零。*nil* 的长度相当于 *self.length*。最后三种形式用 block 的值填充数组。*block* 通过带有被填充的每个元素的绝对索引来传递。
array.first [or] array.first(n)	返回数组的第一个元素或前 *n* 个元素。如果数组为空，则第一种形式返回 *nil*，第二种形式返回一个空的数组。
array.flatten	返回一个新的数组，新数组是一个一维的扁平化的数组（递归）。
array.flatten!	把 *array* 进行扁平化。如果没有变化则返回 *nil*。（数组不包含子数组。）
array.frozen?	如果 *array* 被冻结（或排序时暂时冻结），则返回 true。
array.hash	计算数组的哈希代码。两个具有相同内容的数组将具有相同的哈希代码。
array.include?(obj)	如果 *self* 中包含 *obj*，则返回 true，否则返回 false。
array.index(obj)	返回 *self* 中第一个等于 obj 的对象的 *index*。如果未找到匹配则返回 *nil*。
array.indexes(i1, i2, ... iN) [or] array.indices(i1, i2, ... iN)	该方法在 Ruby 的最新版本中被废弃，所以请使用 Array#values_at。
array.indices(i1, i2, ... iN) [or] array.indexes(i1, i2, ... iN)	该方法在 Ruby 的最新版本中被废弃，所以请使用 Array#values_at。
array.insert(index, obj...)	在给定的 *index* 的元素前插入给定的值，index 可以是负值。
array.inspect	创建一个数组的可打印版本。
array.join(sep=$,)	返回一个字符串，通过把数组的每个元素转换为字符串，并使用 *sep* 分隔进行创建的。
array.last [or] array.last(n)	返回 *self* 的最后一个元素。如果数组为空，则第一种形式返回 *nil*。
array.length	返回 *self* 中元素的个数。可能为零。
array.map { |item| block } [or] array.collect { |item| block }	为 *self* 的每个元素调用一次 *block*。创建一个新的数组，包含 block 返回的值。
array.map! { |item| block }	为 *array* 的每个元素调用一次 *block*，把元素替换为

[or] array.collect! { |item| block }	为 *array* 的每个元素调用一次 *block*，把元素替换为 block 返回的值。
array.nitems	返回 *self* 中 non-nil 元素的个数。可能为零。
array.pack(aTemplateString)	根据 aTemplateString 中的指令，把数组的内容压缩为二进制序列。指令 A、 a 和 Z 后可以跟一个表示结果字段宽度的数字。剩余的指令也可以带有一个表示要转换的数组元素个数的数字。如果数字是一个星号（*），则所有剩余的数组元素都将被转换。任何指令后都可以跟一个下划线（_），表示指定类型使用底层平台的本地尺寸大小，否则使用独立于平台的一致的尺寸大小。在模板字符串中空格会被忽略。
array.pop	从 *array* 中移除最后一个元素，并返回该元素。如果 *array* 为空则返回 *nil*。
array.push(obj, ...)	把给定的 obj 附加到数组的末尾。该表达式返回数组本身，所以几个附加可以连在一起。
array.rassoc(key)	搜索一个数组，其元素也是数组，使用 == 把 *key* 与每个包含的数组的第二个元素进行比较。如果匹配则返回第一个包含的数组。
array.reject { |item| block }	返回一个新的数组，包含当 block 不为 true 时的数组项。
array.reject! { |item| block }	当 block 为真时，从 *array* 删除元素，如果没有变化则返回 *nil*。相当于 Array#delete_if。
array.replace(other_array)	把 *array* 的内容替换为 *other_array* 的内容，必要的时候进行截断或扩充。
array.reverse	返回一个新的数组，包含倒序排列的数组元素。
array.reverse!	把 *array* 进行逆转。
array.reverse_each {|item| block }	与 Array#each 相同，但是把 *array* 进行逆转。
array.rindex(obj)	返回 array 中最后一个等于 obj 的对象的索引。如果未找到匹配，则返回 *nil*。
array.select {|item| block }	调用从数组传入连续元素的 block，返回一个数组，包含 block 返回 *true* 值时的元素。
array.shift	返回 *self* 的第一个元素，并移除该元素（把所有的其他元素下移一位）。如果数组为空，则返回 *nil*。
array.size	返回 *array* 的长度（元素的个数）。length 的别名。
array.slice(index) [or] array.slice(start, length) [or] array.slice(range) [or] array[index] [or] array[start, length] [or] array[range]	返回索引为 *index* 的元素，或者返回从 *start* 开始直至 *length* 个元素的子数组，或者返回 *range* 指定的子数组。负值索引从数组末尾开始计数（-1 是最后一个元素）。如果 *index*（或开始索引）超出范围，则返回 *nil*。
array.slice!(index) [or]	删除 *index*（长度是可选的）或 *range* 指定的元素。返

array.slice!(start, length) [or] array.slice!(range)	回被删除的对象、子数组，如果 *index* 超出范围，则返回 *nil*。
array.sort [or] array.sort { | a,b | block }	返回一个排序的数组。
array.sort! [or] array.sort! { | a,b | block }	把数组进行排序。
array.to_a	返回 *self*。如果在 *Array* 的子类上调用，则把接收参数转换为一个 Array 对象。
array.to_ary	返回 self。
array.to_s	返回 self.join。
array.transpose	假设 self 是数组的数组，且置换行和列。
array.uniq	返回一个新的数组，移除了 *array* 中的重复值。
array.uniq!	从 *self* 中移除重复元素。如果没有变化（也就是说，未找到重复），则返回 *nil*。
array.unshift(obj, ...)	把对象前置在数组的前面，其他元素上移一位。
array.values_at(selector,...)	返回一个数组，包含 self 中与给定的 *selector*（一个或多个）相对应的元素。选择器可以是整数索引或者范围。
array.zip(arg, ...) [or] array.zip(arg, ...){ | arr | block }	把任何参数转换为数组，然后把 *array* 的元素与每个参数中相对应的元素合并。

数组 pack 指令

下表列出了方法 Array#pack 的压缩指令。

指令	描述
@	移动到绝对位置。
A	ASCII 字符串（填充 space，count 是宽度）。
a	ASCII 字符串（填充 null，count 是宽度）。
B	位字符串（降序）
b	位字符串（升序）。
C	无符号字符。
c	字符。
D, d	双精度浮点数，原生格式。
E	双精度浮点数，little-endian 字节顺序。

e	单精度浮点数，little-endian 字节顺序。
F, f	单精度浮点数，原生格式。
G	双精度浮点数，network（big-endian）字节顺序。
g	单精度浮点数，network（big-endian）字节顺序。
H	十六进制字符串（高位优先）。
h	十六进制字符串（低位优先）。
I	无符号整数。
i	整数。
L	无符号 long。
l	Long。
M	引用可打印的，MIME 编码。
m	Base64 编码字符串。
N	Long，network（big-endian）字节顺序。
n	Short，network（big-endian）字节顺序。
P	指向一个结构（固定长度的字符串）。
p	指向一个空结束字符串。
Q, q	64 位数字。
S	无符号 short。
s	Short。
U	UTF-8。
u	UU 编码字符串。
V	Long，little-endian 字节顺序。
v	Short，little-endian 字节顺序。
w	BER 压缩的整数 \fnm。
X	向后跳过一个字节。
x	Null 字节。
Z	与 a 相同，除了 null 会被加上 *。

实例

尝试下面的实例，压缩各种数据。

```
a = [ "a", "b", "c" ]
n = [ 65, 66, 67 ]
puts a.pack("A3A3A3")   #=> "a  b  c  "
puts a.pack("a3a3a3")   #=> "a\000\000b\000\000c\000\000"
puts n.pack("ccc")      #=> "ABC"
```

这将产生以下结果：

```
a  b  c
abc
ABC
```

Ruby 哈希（Hash）

哈希（Hash）是类似 "employee" => "salary" 这样的键值对的集合。哈希的索引是通过任何对象类型的任意键来完成的，而不是一个整数索引，其他与数组相似。

通过键或值遍历哈希的顺序看起来是随意的，且通常不是按照插入顺序。如果您尝试通过一个不存在的键访问哈希，则方法会返回 *nil*。

创建哈希

与数组一样，有各种不同的方式来创建哈希。您可以通过 *new* 类方法创建一个空的哈希：

```
months = Hash.new
```

您也可以使用 *new* 创建带有默认值的哈希，不带默认值的哈希是 *nil*：

```
months = Hash.new( "month" )

or

months = Hash.new "month"
```

当您访问带有默认值的哈希中的任意键时，如果键或值不存在，访问哈希将返回默认值：

```
#!/usr/bin/ruby

months = Hash.new( "month" )

puts "#{months[0]}"
puts "#{months[72]}"
```

这将产生以下结果：

```
month
month
```

```
#!/usr/bin/ruby

H = Hash["a" => 100, "b" => 200]

puts "#{H['a']}"
puts "#{H['b']}"
```

这将产生以下结果：

```
100
200
```

您可以使用任何的 Ruby 对象作为键或值，甚至可以使用数组，所以下面的实例是一个有效的实例：

```
[1,"jan"] => "January"
```

哈希内建方法

我们需要有一个 Hash 对象的实例来调用 Hash 方法。下面是创建 Hash 对象实例的方式：

```
Hash[[key =>|, value]* ] or

Hash.new [or] Hash.new(obj) [or]

Hash.new { |hash, key| block }
```

这将返回一个使用给定对象进行填充的新的哈希。现在，使用创建的对象，我们可以调用任意可用的实例方法。例如：

```
#!/usr/bin/ruby

$, = ", "
months = Hash.new( "month" )

months = {"1" => "January", "2" => "February"}

keys = months.keys

puts "#{keys}"
```

这将产生以下结果：

```
2, 1
```

下面是公共的哈希方法（假设 *hash* 是一个 Hash 对象）：

方法	描述
hash == other_hash	检查两个哈希是否具有相同的键值对个数，键值对是否相互匹配，来判断两个哈希是否相等。
hash.[key]	使用键，从哈希引用值。如果未找到键，则返回默认值。
hash.[key]=value	把 *value* 给定的值与 *key* 给定的键进行关联。
hash.clear	从哈希中移除所有的键值对。
	返回 *hash* 的默认值，如果未通过 default= 进行设置，则

<table>
<tr><td>hash.default(key = nil)</td><td>返回 nil。（如果键在 hash 中不存在，则 [] 返回一个默认值。）</td></tr>
<tr><td>hash.default = obj</td><td>为 hash 设置默认值。</td></tr>
<tr><td>hash.default_proc</td><td>如果 hash 通过块来创建，则返回块。</td></tr>
<tr><td>hash.delete(key) [or]
array.delete(key) { |key| block }</td><td>通过 key 从 hash 中删除键值对。如果使用了块 且未找到匹配的键值对，则返回块的结果。把它与 delete_if 进行比较。</td></tr>
<tr><td>hash.delete_if { |key,value| block }</td><td>为 block 为 true 的每个块，从 hash 中删除键值对。</td></tr>
<tr><td>hash.each { |key,value| block }</td><td>遍历 hash，为每个 key 调用一次 block，传递 key-value 作为一个二元素数组。</td></tr>
<tr><td>hash.each_key { |key| block }</td><td>遍历 hash，为每个 key 调用一次 block，传递 key 作为参数。</td></tr>
<tr><td>hash.each_key { |key_value_array| block }</td><td>遍历 hash，为每个 key 调用一次 block，传递 key 和 value 作为参数。</td></tr>
<tr><td>hash.each_key { |value| block }</td><td>遍历 hash，为每个 key 调用一次 block，传递 value 作为参数。</td></tr>
<tr><td>hash.empty?</td><td>检查 hash 是否为空（不包含键值对），返回 true 或 false。</td></tr>
<tr><td>hash.fetch(key [, default]) [or] hash.fetch(key) { | key | block }</td><td>通过给定的 key 从 hash 返回值。如果未找到 key，且未提供其他参数，则抛出 IndexError 异常；如果给出了 default，则返回 default；如果指定了可选的 block，则返回 block 的结果。</td></tr>
<tr><td>hash.has_key?(key) [or]
hash.include?(key) [or]
hash.key?(key) [or]
hash.member?(key)</td><td>检查给定的 key 是否存在于哈希中，返回 true 或 false。</td></tr>
<tr><td>hash.has_value?(value)</td><td>检查哈希是否包含给定的 value。</td></tr>
<tr><td>hash.index(value)</td><td>为给定的 value 返回哈希中的 key，如果未找到匹配值则返回 nil。</td></tr>
<tr><td>hash.indexes(keys)</td><td>返回一个新的数组，由给定的键的值组成。找不到的键将插入默认值。该方法已被废弃，请使用 select。</td></tr>
<tr><td>hash.indices(keys)</td><td>返回一个新的数组，由给定的键的值组成。找不到的键将插入默认值。该方法已被废弃，请使用 select。</td></tr>
<tr><td>hash.inspect</td><td>返回哈希的打印字符串版本。</td></tr>
<tr><td>hash.invert</td><td>创建一个新的 hash，倒置 hash 中的 keys 和 values。也就是说，在新的哈希中，hash 中的键将变成值，值将变成键。</td></tr>
<tr><td>hash.keys</td><td>创建一个新的数组，带有 hash 中的键。/td></td></tr>
<tr><td>hash.length</td><td>以整数形式返回 hash 的大小或长度。</td></tr>
</table>

hash.merge(other_hash) [or] hash.merge(other_hash) { |key, oldval, newval| block }	返回一个新的哈希，包含 *hash* 和 *other_hash* 的内容，重写 hash 中与 *other_hash* 带有重复键的键值对。
hash.merge!(other_hash) [or] hash.merge!(other_hash) { |key, oldval, newval| block }	与 merge 相同，但实际上 hash 发生了变化。
hash.rehash	基于每个 *key* 的当前值重新建立 *hash*。如果插入后值发生了改变，该方法会重新索引 *hash*。
hash.reject { |key, value| block }	为 *block* 为 *true* 的每个键值对创建一个新的 *hash*。
hash.reject! { |key, value| block }	与 *reject* 相同，但实际上 hash 发生了变化。
hash.replace(other_hash)	把 *hash* 的内容替换为 *other_hash* 的内容。
hash.select { |key, value| block }	返回一个新的数组，由 *block* 返回 *true* 的 *hash* 中的键值对组成。
hash.shift	从 *hash* 中移除一个键值对，并把该键值对作为二元素数组返回。
hash.size	以整数形式返回 *hash* 的 *size* 或 length。
hash.sort	把 *hash* 转换为一个包含键值对数组的二维数组，然后进行排序。
hash.store(key, value)	存储 *hash* 中的一个键值对。
hash.to_a	从 hash 中创建一个二维数组。每个键值对转换为一个数组，所有这些数组都存储在一个数组中。
hash.to_hash	返回 *hash*（self）。
hash.to_s	把 *hash* 转换为一个数组，然后把该数组转换为一个字符串。
hash.update(other_hash) [or] hash.update(other_hash) {|key, oldval, newval| block}	返回一个新的哈希，包含 *hash* 和 *other_hash* 的内容，重写 *hash* 中与 *other_hash* 带有重复键的键值对。
hash.value?(value)	检查 *hash* 是否包含给定的 *value*。
hash.values	返回一个新的数组，包含 *hash* 的所有值。
hash.values_at(obj, ...)	返回一个新的数组，包含 *hash* 中与给定的键相关的值。

Ruby 日期 & 时间（Date & Time）

Time 类在 Ruby 中用于表示日期和时间。它是基于操作系统提供的系统日期和时间之上。该类可能无法表示 1970 年之前或者 2038 年之后的日期。

本教程将让您熟悉日期和时间的所有重要的概念。

创建当前的日期和时间

下面是获取当前的日期和时间的简单实例：

```
#!/usr/bin/ruby -w

time1 = Time.new

puts "Current Time : " + time1.inspect

# Time.now 是一个同义词
time2 = Time.now
puts "Current Time : " + time2.inspect
```

这将产生以下结果：

```
Current Time : Mon Jun 02 12:02:39 -0700 2008
Current Time : Mon Jun 02 12:02:39 -0700 2008
```

获取 Date & Time 组件

我们可以使用 *Time* 对象来获取各种日期和时间的组件。请看下面的实例：

```
#!/usr/bin/ruby -w

time = Time.new

# Time 的组件
puts "Current Time : " + time.inspect
puts time.year    # => 日期的年份
puts time.month   # => 日期的月份（1 到 12）
puts time.day     # => 一个月中的第几天（1 到 31）
puts time.wday    # => 一周中的星期几（0 是星期日）
puts time.yday    # => 365：一年中的第几天
puts time.hour    # => 23：24 小时制
puts time.min     # => 59
puts time.sec     # => 59
puts time.usec    # => 999999：微秒
puts time.zone    # => "UTC"：时区名称
```

这将产生以下结果：

```
Current Time : Mon Jun 02 12:03:08 -0700 2008
2008
6
2
1
154
12
3
8
247476
UTC
```

Time.utc、*Time.gm* 和 *Time.local* 函数

这些函数可用于格式化标准格式的日期，如下所示：

```
# July 8, 2008
Time.local(2008, 7, 8)
# July 8, 2008, 09:10am，本地时间
Time.local(2008, 7, 8, 9, 10)
# July 8, 2008, 09:10 UTC
Time.utc(2008, 7, 8, 9, 10)
# July 8, 2008, 09:10:11 GMT （与 UTC 相同）
Time.gm(2008, 7, 8, 9, 10, 11)
```

下面的实例在数组中获取所有的组件：

```
[sec,min,hour,day,month,year,wday,yday,isdst,zone]
```

尝试下面的实例：

```
#!/usr/bin/ruby -w

time = Time.new

values = time.to_a
p values
```

这将产生以下结果：

```
[26, 10, 12, 2, 6, 2008, 1, 154, false, "MST"]
```

该数组可被传到 *Time.utc* 或 *Time.local* 函数来获取日期的不同格式，如下所示：

```
#!/usr/bin/ruby -w

time = Time.new

values = time.to_a
puts Time.utc(*values)
```

这将产生以下结果：

```
Mon Jun 02 12:15:36 UTC 2008
```

下面是获取时间的方式，从纪元以来的秒数（平台相关）：

```
# 返回从纪元以来的秒数
time = Time.now.to_i

# 把秒数转换为 Time 对象
Time.at(time)

# 返回从纪元以来的秒数，包含微妙
time = Time.now.to_f
```

时区和夏令时

您可以使用 *Time* 对象来获取与时区和夏令时有关的所有信息，如下所示：

```
time = Time.new

# 这里是解释
time.zone       # => "UTC"：返回时区
time.utc_offset # => 0：UTC 是相对于 UTC 的 0 秒偏移
time.zone       # => "PST"（或其他时区）
time.isdst      # => false：如果 UTC 没有 DST（夏令时）
time.utc?       # => true：如果在 UTC 时区
time.localtime  # 转换为本地时区
time.gmtime     # 转换回 UTC
time.getlocal   # 返回本地区中的一个新的 Time 对象
time.getutc     # 返回 UTC 中的一个新的 Time 对象
```

格式化时间和日期

有多种方式格式化日期和时间。下面的实例演示了其中一部分：

```
#!/usr/bin/ruby -w
time = Time.new

puts time.to_s
puts time.ctime
puts time.localtime
puts time.strftime("%Y-%m-%d %H:%M:%S")
```

这将产生以下结果：

```
Mon Jun 02 12:35:19 -0700 2008
Mon Jun  2 12:35:19 2008
Mon Jun 02 12:35:19 -0700 2008
2008-06-02 12:35:19
```

时间格式化指令

下表所列出的指令与方法 *Time.strftime* 一起使用。

指令	描述
%a	星期几名称的缩写（比如 Sun）。
%A	星期几名称的全称（比如 Sunday）。
%b	月份名称的缩写（比如 Jan）。
%B	月份名称的全称（比如 January）。
%c	优选的本地日期和时间表示法。
%d	一个月中的第几天（01 到 31）。
%H	一天中的第几小时，24 小时制（00 到 23）。
%I	一天中的第几小时，12 小时制（01 到 12）。
%j	一年中的第几天（001 到 366）。
%m	一年中的第几月（01 到 12）。
%M	小时中的第几分钟（00 到 59）。
%p	子午线指示（AM 或 PM）。
%S	分钟中的第几秒（00 或 60）。
%U	当前年中的周数，从第一个星期日（作为第一周的第一天）开始（00 到 53）。
%W	当前年中的周数，从第一个星期一（作为第一周的第一天）开始（00 到 53）。
%w	一星期中的第几天（Sunday 是 0，0 到 6）。
%x	只有日期没有时间的优先表示法。
%X	只有时间没有日期的优先表示法。
%y	不带世纪的年份表示（00 到 99）。
%Y	带有世纪的年份。
%Z	时区名称。
%%	% 字符。

时间算法

您可以用时间做一些简单的算术，如下所示：

```
now = Time.now           # 当前时间
puts now

past = now - 10          # 10 秒之前。Time - number => Time
puts past

future = now + 10        # 从现在开始 10 秒之后。Time + number => Time
puts future

diff = future - now      # => 10  Time - Time => 秒数
puts diff
```

这将产生以下结果：

```
Thu Aug 01 20:57:05 -0700 2013
Thu Aug 01 20:56:55 -0700 2013
Thu Aug 01 20:57:15 -0700 2013
10.0
```

Ruby 范围（Range）

范围（Range）无处不在：January 到 December、 0 到 9、等等。Ruby 支持范围，并允许我们以不同的方式使用范围：

- 作为序列的范围
- 作为条件的范围
- 作为间隔的范围

作为序列的范围

范围的第一个也是最常见的用途是表达序列。序列有一个起点、一个终点和一个在序列产生连续值的方式。

Ruby 使用 **".."** 和 **"..."** 范围运算符创建这些序列。两点形式创建一个包含指定的最高值的范围，三点形式创建一个不包含指定的最高值的范围。

```
(1..5)        #==> 1, 2, 3, 4, 5
(1...5)       #==> 1, 2, 3, 4
('a'..'d')    #==> 'a', 'b', 'c', 'd'
```

序列 1..100 是一个 *Range* 对象，包含了两个 *Fixnum* 对象的引用。如果需要，您可以使用 *to_a* 方法把范围转换为列表。尝试下面的实例：

```
#!/usr/bin/ruby

$, =", "   # Array 值分隔符
range1 = (1..10).to_a
range2 = ('bar'..'bat').to_a

puts "#{range1}"
puts "#{range2}"
```

这将产生以下结果：

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
bar, bas, bat
```

范围实现了让您可以遍历它们的方法，您可以通过多种方式检查它们的内容：

```
#!/usr/bin/ruby

# Assume a range
digits = 0..9

puts digits.include?(5)
ret = digits.min
puts "Min value is #{ret}"

ret = digits.max
puts "Max value is #{ret}"

ret = digits.reject {|i| i < 5 }
puts "Rejected values are #{ret}"

digits.each do |digit|
   puts "In Loop #{digit}"
end
```

这将产生以下结果：

```
true
Min value is 0
Max value is 9
Rejected values are 5, 6, 7, 8, 9
In Loop 0
In Loop 1
In Loop 2
In Loop 3
In Loop 4
In Loop 5
In Loop 6
In Loop 7
In Loop 8
In Loop 9
```

作为条件的范围

范围也可以用作条件表达式。例如，下面的代码片段从标准输入打印行，其中每个集合的第一行包含单词 *start*，最后一行包含单词 *end.*：

```
while gets
   print if /start/../end/
end
```

范围可以用在 case 语句中：

```
#!/usr/bin/ruby

score = 70

result = case score
   when 0..40: "Fail"
   when 41..60: "Pass"
   when 61..70: "Pass with Merit"
   when 71..100: "Pass with Distinction"
   else "Invalid Score"
end

puts result
```

这将产生以下结果：

```
Pass with Merit
```

作为间隔的范围

范围的最后一个用途是间隔测试：检查某些值是否落在范围表示的间隔里。这是使用 === 相等运算符来完成的。

```
#!/usr/bin/ruby

if ((1..10) === 5)
  puts "5 lies in (1..10)"
end

if (('a'..'j') === 'c')
  puts "c lies in ('a'..'j')"
end

if (('a'..'j') === 'z')
  puts "z lies in ('a'..'j')"
end
```

这将产生以下结果：

```
5 lies in (1..10)
c lies in ('a'..'j')
```

Ruby 迭代器

迭代器是集合支持的方法。存储一组数据成员的对象称为集合。在 Ruby 中，数组和散列可以称之为集合。

迭代器返回集合的所有元素，一个接着一个。在这里我们将讨论两种迭代器，*each* 和 *collect*。

Ruby *each* 迭代器

each 迭代器返回数组或哈希的所有元素。

语法

```
collection.each do |variable|
   code
end
```

为集合中的每个元素执行 *code*。在这里，集合可以是数组或哈希。

实例

```
#!/usr/bin/ruby

ary = [1,2,3,4,5]
ary.each do |i|
   puts i
end
```

这将产生以下结果：

```
1
2
3
4
5
```

each 迭代器总是与一个块关联。它向块返回数组的每个值，一个接着一个。值被存储在变量 **i** 中，然后显示在屏幕上。

Ruby *collect* 迭代器

collect 迭代器返回集合的所有元素。

语法

```
collection = collection.collect
```

collect 方法不需要总是与一个块关联。*collect* 方法返回整个集合，不管它是数组或者是哈希。

实例

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = Array.new
b = a.collect
puts b
```

这将产生以下结果：

```
1
2
3
4
5
```

注意：*collect* 方法不是数组间进行复制的正确方式。这里有另一个称为 *clone* 的方法，用于复制一个数组到另一个数组。

当您想要对每个值进行一些操作以便获得新的数组时，您通常使用 collect 方法。例如，下面的代码会生成一个数组，其值是 a 中每个值的 10 倍。

```
#!/usr/bin/ruby

a = [1,2,3,4,5]
b = a.collect{|x| 10*x}
puts b
```

这将产生以下结果：

```
10
20
30
40
50
```

Ruby 文件的输入与输出

Ruby 提供了一整套 I/O 相关的方法，在内核（Kernel）模块中实现。所有的 I/O 方法派生自 IO 类。

类 *IO* 提供了所有基础的方法，比如 *read*、 *write*、 *gets*、 *puts*、 *readline*、 *getc* 和 *printf*。

本章节将讲解所有 Ruby 中可用的基础的 I/O 函数。如需了解更多的函数，请查看 Ruby 的 *IO* 类。

puts 语句

在前面的章节中，您赋值给变量，然后使用 *puts* 语句打印输出。

puts 语句指示程序显示存储在变量中的值。这将在每行末尾添加一个新行。

实例

```
#!/usr/bin/ruby

val1 = "This is variable one"
val2 = "This is variable two"
puts val1
puts val2
```

这将产生以下结果：

```
This is variable one
This is variable two
```

gets 语句

gets 语句可用于获取来自名为 STDIN 的标准屏幕的用户输入。

实例

下面的代码演示了如何使用 gets 语句。该代码将提示用户输入一个值，该值将被存储在变量 val 中，最后会被打印在 STDOUT 上。

```
#!/usr/bin/ruby

puts "Enter a value :"
val = gets
puts val
```

这将产生以下结果：

```
Enter a value :
This is entered value
This is entered value
```

putc 语句

与 *puts* 语句不同，*puts* 语句输出整个字符串到屏幕上，而 *putc* 语句可用于依次输出一个字符。

实例

下面代码的输出只是字符 H：

```
#!/usr/bin/ruby

str="Hello Ruby!"
putc str
```

这将产生以下结果：

```
H
```

print 语句

print 语句与 *puts* 语句类似。唯一的不同在于 *puts* 语句在输出内容后会跳到下一行，而使用 *print* 语句时，光标定位在同一行。

实例

```
#!/usr/bin/ruby

print "Hello World"
print "Good Morning"
```

这将产生以下结果：

```
Hello WorldGood Morning
```

打开和关闭文件

截至现在，您已经读取并写入标准输入和输出。现在，我们将看看如何操作实际的数据文件。

File.new 方法

您可以使用 *File.new* 方法创建一个 *File* 对象用于读取、写入或者读写，读写权限取决于 mode 字符串。最后，您可以使用 *File.close* 方法来关闭该文件。

语法

```
aFile = File.new("filename", "mode")
   # ... 处理文件
aFile.close
```

File.open 方法

您可以使用 *File.open* 方法创建一个新的 file 对象，并把该 file 对象赋值给文件。但是，*File.open* 和 *File.new* 方法之间有一点不同。不同点是 *File.open* 方法可与块关联，而 *File.new* 方法不能。

```
File.open("filename", "mode") do |aFile|
   # ... process the file
end
```

下表列出了打开文件的不同模式：

模式	描述
r	只读模式。文件指针被放置在文件的开头。这是默认模式。
r+	读写模式。文件指针被放置在文件的开头。
w	只写模式。如果文件存在，则重写文件。如果文件不存在，则创建一个新文件用于写入。
w+	读写模式。如果文件存在，则重写已存在的文件。如果文件不存在，则创建一个新文件用于读写。
a	只写模式。如果文件存在，则文件指针被放置在文件的末尾。也就是说，文件是追加模式。如果文件不存在，则创建一个新文件用于写入。
a+	读写模式。如果文件存在，则文件指针被放置在文件的末尾。也就是说，文件是追加模式。如果文件不存在，则创建一个新文件用于读写。

读取和写入文件

用于简单 I/O 的方法也可用于所有 file 对象。所以，gets 从标准输入读取一行，*aFile.gets* 从文件对象 aFile 读取一行。

但是，I/O 对象提供了访问方法的附加设置，为我们提供了便利。

sysread 方法

您可以使用方法 *sysread* 来读取文件的内容。当使用方法 sysread 时，您可以使用任意一种模式打开文件。例如：

下面是输入文本文件：

```
This is a simple text file for testing purpose.
```

现在让我们尝试读取这个文件：

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r")
if aFile
   content = aFile.sysread(20)
   puts content
else
   puts "Unable to open file!"
end
```

该语句将输入文件的头 20 个字符。文件指针将被放置在文件中第 21 个字符的位置。

syswrite 方法

您可以使用方法 *syswrite* 来向文件写入内容。当使用方法 syswrite 时，您需要以写入模式打开文件。例如：

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
else
   puts "Unable to open file!"
end
```

该语句将写入 "ABCDEF" 到文件中。

each_byte 方法

该方法属于类 *File*。方法 *each_byte* 总是与块相关联。请看下面的代码实例：

```
#!/usr/bin/ruby

aFile = File.new("input.txt", "r+")
if aFile
   aFile.syswrite("ABCDEF")
   aFile.each_byte {|ch| putc ch; putc ?. }
else
   puts "Unable to open file!"
end
```

字符一个接着一个被传到变量 ch，然后显示在屏幕上，如下所示：

```
s. .a. .s.i.m.p.l.e. .t.e.x.t. .f.i.l.e. .f.o.r. .t.e.s.t.i.n.g. .p.u.r.p.o.s.e...
.
.
```

IO.readlines 方法

类 *File* 是类 IO 的一个子类。类 IO 也有一些用于操作文件的方法。

IO.readlines 是 IO 类中的一个方法。该方法逐行返回文件的内容。下面的代码显示了方法 *IO.readlines* 的使用：

```
#!/usr/bin/ruby

arr = IO.readlines("input.txt")
puts arr[0]
puts arr[1]
```

在这段代码中，变量 arr 是一个数组。文件 *input.txt* 的每一行将是数组 arr 中的一个元素。因此，arr[0] 将包含第一行，而 arr[1] 将包含文件的第二行。

IO.foreach 方法

该方法也逐行返回输出。方法 *foreach* 与方法 *readlines* 之间不同的是，方法 *foreach* 与块相关联。但是，不像方法 *readlines*，方法 *foreach* 不是返回一个数组。例如：

```
#!/usr/bin/ruby

IO.foreach("input.txt"){|block| puts block}
```

这段代码将把文件 *test* 的内容逐行传给变量 block，然后输出将显示在屏幕上。

重命名和删除文件

您可以通过 *rename* 和 *delete* 方法重命名和删除文件。

下面的实例重命名一个已存在文件 *test1.txt*：

```
#!/usr/bin/ruby

# 重命名文件 test1.txt 为 test2.txt
File.rename( "test1.txt", "test2.txt" )
```

下面的实例删除一个已存在文件 *test2.txt*：

```
#!/usr/bin/ruby

# 删除文件 test2.txt
File.delete("text2.txt")
```

文件模式与所有权

使用带有掩码的 *chmod* 方法来改变文件的模式或权限/访问列表：

下面的实例改变一个已存在文件 *test.txt* 的模式为一个掩码值：

```
#!/usr/bin/ruby

file = File.new( "test.txt", "w" )
file.chmod( 0755 )
```

下表列出了 *chmod* 方法中可使用的不同的掩码：

掩码	描述
0700	rwx 掩码，针对所有者
0400	r，针对所有者
0200	w，针对所有者
0100	x，针对所有者
0070	rwx 掩码，针对所属组
0040	r，针对所属组
0020	w，针对所属组
0010	x，针对所属组
0007	rwx 掩码，针对其他人
0004	r，针对其他人
0002	w，针对其他人
0001	x，针对其他人
4000	执行时设置用户 ID
2000	执行时设置所属组 ID
1000	保存交换文本，甚至在使用后也会保存

文件查询

下面的命令在打开文件前检查文件是否已存在：

```
#!/usr/bin/ruby

File.open("file.rb") if File::exists?( "file.rb" )
```

下面的命令查询文件是否确实是一个文件：

```
#!/usr/bin/ruby

# 返回 <i>true</i> 或 <i>false</i>
File.file?( "text.txt" )
```

下面的命令检查给定的文件名是否是一个目录：

```
#!/usr/bin/ruby

# 一个目录
File::directory?( "/usr/local/bin" ) # => true

# 一个文件
File::directory?( "file.rb" ) # => false
```

下面的命令检查文件是否可读、可写、可执行：

```
#!/usr/bin/ruby

File.readable?( "test.txt" )   # => true
File.writable?( "test.txt" )   # => true
File.executable?( "test.txt" ) # => false
```

下面的命令检查文件是否大小为零：

```
#!/usr/bin/ruby

File.zero?( "test.txt" )      # => true
```

下面的命令返回文件的大小：

```
#!/usr/bin/ruby

File.size?( "text.txt" )     # => 1002
```

下面的命令用于检查文件的类型：

```
#!/usr/bin/ruby

File::ftype( "test.txt" )     # => file
```

ftype 方法通过返回下列中的某个值来标识了文件的类型：*file*、 *directory*、 *characterSpecial*、 *blockSpecial*、 *fifo*、 *link*、 *socket* 或 *unknown*。

下面的命令用于检查文件被创建、修改或最后访问的时间：

```
#!/usr/bin/ruby

File::ctime( "test.txt" ) # => Fri May 09 10:06:37 -0700 2008
File::mtime( "text.txt" ) # => Fri May 09 10:44:44 -0700 2008
File::atime( "text.txt" ) # => Fri May 09 10:45:01 -0700 2008
```

Ruby 中的目录

所有的文件都是包含在目录中，Ruby 提供了处理文件和目录的方式。*File* 类用于处理文件，*Dir* 类用于处理目录。

浏览目录

为了在 Ruby 程序中改变目录，请使用 *Dir.chdir*。下面的实例改变当前目录为 */usr/bin*。

```
Dir.chdir("/usr/bin")
```

您可以通过 *Dir.pwd* 查看当前目录：

```
puts Dir.pwd # 返回当前目录，类似 /usr/bin
```

您可以使用 *Dir.entries* 获取指定目录内的文件和目录列表：

```
puts Dir.entries("/usr/bin").join(' ')
```

Dir.entries 返回一个数组，包含指定目录内的所有项。*Dir.foreach* 提供了相同的功能：

```
Dir.foreach("/usr/bin") do |entry|
   puts entry
end
```

获取目录列表的一个更简洁的方式是通过使用 Dir 的类数组的方法：

```
Dir["/usr/bin/*"]
```

创建目录

Dir.mkdir 可用于创建目录：

```
Dir.mkdir("mynewdir")
```

您也可以通过 mkdir 在新目录（不是已存在的目录）上设置权限：

注意：掩码 755 设置所有者（owner）、所属组（group）、每个人（world [anyone]）的权限为 rwxr-xr-x，其中 r = read 读取，w = write 写入，x = execute 执行。

```
Dir.mkdir( "mynewdir", 755 )
```

删除目录

Dir.delete 可用于删除目录。*Dir.unlink* 和 *Dir.rmdir* 执行同样的功能，为我们提供了便利。

```
Dir.delete("testdir")
```

创建文件 & 临时目录

临时文件是那些在程序执行过程中被简单地创建，但不会永久性存储的信息。

Dir.tmpdir 提供了当前系统上临时目录的路径，但是该方法默认情况下是不可用的。为了让 *Dir.tmpdir* 可用，使用必需的 'tmpdir' 是必要的。

您可以把 *Dir.tmpdir* 和 *File.join* 一起使用，来创建一个独立于平台的临时文件：

```
require 'tmpdir'
   tempfilename = File.join(Dir.tmpdir, "tingtong")
   tempfile = File.new(tempfilename, "w")
   tempfile.puts "This is a temporary file"
   tempfile.close
   File.delete(tempfilename)
```

这段代码创建了一个临时文件，并向其中写入数据，然后删除文件。Ruby 的标准库也包含了一个名为 *Tempfile* 的库，该库可用于创建临时文件：

```
require 'tempfile'
   f = Tempfile.new('tingtong')
   f.puts "Hello"
   puts f.path
   f.close
```

内建函数

下面提供了 Ruby 中处理文件和目录的内建函数的完整列表：

- File 类和方法。
- Dir 类和方法。

Ruby File 类和方法

File 表示一个连接到普通文件的 *stdio* 对象。open 为普通文件返回该类的一个实例。

类方法

方法	描述
File::atime(path)	返回 *path* 的最后访问时间。
File::basename(path[, suffix])	返回 *path* 末尾的文件名。如果指定了 *suffix*，则它会从文件名末尾被删除。 例如：File.basename("/home/users/bin/ruby.exe") #=> "ruby.exe"
File::blockdev?(path)	如果 path 是一个块设备，则返回 true。
File::chardev?(path)	如果 path 是一个字符设备，则返回 true。
File::chmod(mode, path...)	改变指定文件的权限模式。
File::chown(owner, group, path...)	改变指定文件的所有者和所属组。
File::ctime(path)	返回 path 的最后一个 inode 更改时间。
File::delete(path...) **File::unlink(path...)**	删除指定的文件。
File::directory?(path)	如果 path 是一个目录，则返回 true。
File::dirname(path)	返回 path 的目录部分，不包括最后的文件名。
File::executable?(path)	如果 path 是可执行的，则返回 true。
File::executable_real?(path)	如果 path 通过真正的用户权限是可执行的，则返回 true。
File::exist?(path)	如果 path 存在，则返回 true。
File::expand_path(path[, dir])	返回 path 的绝对路径，扩展 ~ 为进程所有者的主目录，~user 为用户的主目录。相对路径是相对于 dir 指定的目录，如果 dir 被省略则相对于当前工作目录。
File::file?(path)	如果 path 是一个普通文件，则返回 true。
File::ftype(path)	返回下列其中一个字符串，表示文件类型： **file** - 普通文件 **directory** - 目录 **characterSpecial** - 字符特殊文件 **blockSpecial** - 块特殊文件 **fifo** - 命名管道（FIFO） **link** - 符号链接 **socket** - Socket **unknown** - 未知的文件类型
File::grpowned?(path)	如果 path 由用户的所属组所有，则返回 true。

File::join(item...)	返回一个字符串，由指定的项连接在一起，并使用 File::Separator 进行分隔。 例如：File::join("", "home", "usrs", "bin") # => "/home/usrs/bin"
File::link(old, new)	创建一个到文件 old 的硬链接。
File::lstat(path)	与 stat 相同，但是它返回自身符号链接上的信息，而不是所指向的文件。
File::mtime(path)	返回 path 的最后一次修改时间。
File::new(path[, mode="r"]) File::open(path[, mode="r"]) File::open(path[, mode="r"]) {|f| ...}	打开文件。如果指定了块，则通过传递新文件作为参数来执行块。当块退出时，文件会自动关闭。这些方法有别于 Kernel.open，即使 path 是以 | 开头，后续的字符串也不会作为命令运行。
File::owned?(path)	如果 path 由有效的用户所有，则返回 true。
File::pipe?(path)	如果 path 是一个管道，则返回 true。
File::readable?(path)	如果 path 是可读的，则返回 true。
File::readable_real?(path)	如果 path 通过真正的用户权限是可读的，则返回 true。
File::readlink(path)	返回 path 所指向的文件。
File::rename(old, new)	改变文件名 old 为 new。
File::setgid?(path)	如果设置了 path 的 set-group-id 权限位，则返回 true。
File::setuid?(path)	如果设置了 path 的 set-user-id 权限位，则返回 true。
File::size(path)	返回 path 的文件大小。
File::size?(path)	返回 path 的文件大小，如果为 0 则返回 nil。
File::socket?(path)	如果 path 是一个 socket，则返回 true。
File::split(path)	返回一个数组，包含 path 的内容，path 被分成 File::dirname(path) 和 File::basename(path)。
File::stat(path)	返回 path 上带有信息的 File::Stat 对象。
File::sticky?(path)	如果设置了 path 的 sticky 位，则返回 true。
File::symlink(old, new)	创建一个指向文件 old 的符号链接。
File::symlink?(path)	如果 path 是一个符号链接，则返回 true。
File::truncate(path, len)	截断指定的文件为 len 字节。
File::unlink(path...)	删除 path 给定的文件。
File::umask([mask])	如果未指定参数，则为该进程返回当前的 umask。如果指定了一个参数，则设置了 umask，并返回旧的 umask。
File::utime(atime, mtime, path...)	改变指定文件的访问和修改时间。

File::writable?(path)	如果 path 是可写的，则返回 true。
File::writable_real?(path)	如果 path 通过真正的用户权限是可写的，则返回 true。
File::zero?(path)	如果 path 的文件大小是 0，则返回 true。

实例方法

假设 **f** 是 **File** 类的一个实例：

方法	描述
f.atime	返回 f 的最后访问时间。
f.chmode(mode)	改变 f 的权限模式。
f.chown(owner, group)	改变 f 的所有者和所属组。
f.ctime	返回 f 的最后一个 inode 更改时间。
f.flock(op)	调用 flock(2)。op 可以是 0 或一个逻辑值或 File 类常量 LOCK_EX、LOCK_NB、LOCK_SH 和 LOCK_UN。
f.lstat	与 stat 相同，但是它返回自身符号链接上的信息，而不是所指向的文件。
f.mtime	返回 f 的最后修改时间。
f.path	返回用于创建 f 的路径名。
f.reopen(path[, mode="r"])	重新打开文件。
f.truncate(len)	截断 f 为 len 字节。

Ruby Dir 类和方法

Dir | 是一个表示用于给出操作系统中目录中的文件名的目录流。Dir 类也拥有与目录相关的操作，比如通配符文件名匹配、改变工作目录等。

类方法

方法	描述
Dir[pat] Dir::glob(pat)	返回一个数组，包含与指定的通配符模式 pat 匹配的文件名： ***** - 匹配包含 null 字符串的任意字符串 ****** - 递归地匹配任意字符串 **?** - 匹配任意单个字符 **[...]** - 匹配封闭字符中的任意一个 **{a,b...}** - 匹配字符串中的任意一个 Dir["foo.*"] # 匹配 "foo.c"、 "foo.rb" 等等 Dir["foo.?"] # 匹配 "foo.c"、 "foo.h" 等等
Dir::chdir(path)	改变当前目录。
Dir::chroot(path)	改变根目录（只允许超级用户）。并不是在所有的平台上都可用。
Dir::delete(path)	删除 path 指定的目录。目录必须是空的。
Dir::entries(path)	返回一个数组，包含目录 path 中的文件名。
Dir::foreach(path) {| f| ...}	为 path 指定的目录中的每个文件执行一次块。
Dir::getwd Dir::pwd	返回当前目录。
Dir::mkdir(path[, mode=0777])	创建 path 指定的目录。权限模式可被 File::umask 的值修改，在 Win32 的平台上会被忽略。
Dir::new(path) Dir::open(path) Dir::open(path) {| dir| ...}	返回 path 的新目录对象。如果 open 给出一个块，则新目录对象会传到该块，块会在终止前关闭目录对象。
Dir::pwd	参见 Dir::getwd。
Dir::rmdir(path) Dir::unlink(path) Dir::delete(path)	删除 path 指定的目录。目录必须是空的。

实例方法

假设 **d** 是 **\|Dir** 类的一个实例：

方法	描述
d.close	关闭目录流。
d.each {| f| ...}	为 d 中的每一个条目执行一次块。
d.pos	d.tell 返回 d 中的当前位置。
d.pos= offset	设置目录流中的位置。
d.pos= pos **d.seek(pos)**	移动到 d 中的某个位置。pos 必须是一个由 d.pos 返回的值或 0。
d.read	返回 d 的下一个条目。
d.rewind	移动 d 中的位置到第一个条目。
d.seek(po s)	参见 d.pos=pos。
d.tell	参见 d.pos。

Ruby 异常

异常和执行总是被联系在一起。如果您打开一个不存在的文件，且没有恰当地处理这种情况，那么您的程序则被认为是低质量的。

如果异常发生，则程序停止。异常用于处理各种类型的错误，这些错误可能在程序执行期间发生，所以要采取适当的行动，而不至于让程序完全停止。

Ruby 提供了一个完美的处理异常的机制。我们可以在 *begin/end* 块中附上可能抛出异常的代码，并使用 *rescue* 子句告诉 Ruby 完美要处理的异常类型。

语法

```
begin
# -
rescue OneTypeOfException
# -
rescue AnotherTypeOfException
# -
else
# 其他异常
ensure
# 总是被执行
end
```

从 *begin* 到 *rescue* 中的一切是受保护的。如果代码块执行期间发生了异常，控制会传到 *rescue* 和 *end* 之间的块。

对于 *begin* 块中的每个 *rescue* 子句，Ruby 把抛出的异常与每个参数进行轮流比较。如果 rescue 子句中命名的异常与当前抛出的异常类型相同，或者是该异常的父类，则匹配成功。

如果异常不匹配所有指定的错误类型，我们可以在所有的 *rescue* 子句后使用一个 *else* 子句。

实例

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
      file = STDIN
end
print file, "==", STDIN, "\n"
```

这将产生以下结果。您可以看到，*STDIN* 取代了 *file*，因为打开失败。

```
#<IO:0xb7d16f84>==#<IO:0xb7d16f84>
```

使用 *retry* 语句

您可以使用 *rescue* 块捕获异常，然后使用 *retry* 语句从开头开始执行 *begin* 块。

语法

```
begin
    # 这段代码抛出的异常将被下面的 rescue 子句捕获
rescue
    # 这个块将捕获所有类型的异常
    retry  # 这将把控制移到 begin 的开头
end
```

实例

```
#!/usr/bin/ruby

begin
   file = open("/unexistant_file")
   if file
      puts "File opened successfully"
   end
rescue
   fname = "existant_file"
   retry
end
```

以下是处理流程：

- 打开时发生异常。
- 跳到 rescue。fname 被重新赋值。
- 通过 retry 跳到 begin 的开头。
- 这次文件成功打开。
- 继续基本的过程。

注意：如果被重新命名的文件不存在，本势力代码会无限尝试。所以异常处理时，谨慎使用 *retry*。

使用 *raise* 语句

您可以使用 *raise* 语句抛出异常。下面的方法在调用时抛出异常。它的第二个消息将被输出。

语法

```
raise

OR

raise "Error Message"

OR

raise ExceptionType, "Error Message"

OR

raise ExceptionType, "Error Message" condition
```

第一种形式简单地重新抛出当前异常（如果没有当前异常则抛出一个 RuntimeError）。这用在传入异常之前需要解释异常的异常处理程序中。

第二种形式创建一个新的 *RuntimeError* 异常，设置它的消息为给定的字符串。该异常之后抛出到调用堆栈。

第三种形式使用第一个参数创建一个异常，然后设置相关的消息为第二个参数。

第四种形式与第三种形式类似，您可以添加任何额外的条件语句（比如 *unless*）来抛出异常。

实例

```
#!/usr/bin/ruby

begin
    puts 'I am before the raise.'
    raise 'An error has occurred.'
    puts 'I am after the raise.'
rescue
    puts 'I am rescued.'
end
puts 'I am after the begin block.'
```

这将产生以下结果：

```
I am before the raise.
I am rescued.
I am after the begin block.
```

另一个演示 *raise* 用法的实例：

```
#!/usr/bin/ruby

begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
end
```

这将产生以下结果：

```
A test exception.
["main.rb:4"]
```

使用 *ensure* 语句

有时候，无论是否抛出异常，您需要保证一些处理在代码块结束时完成。例如，您可能在进入时打开了一个文件，当您退出块时，您需要确保关闭文件。

ensure 子句做的就是这个。ensure 放在最后一个 rescue 子句后，并包含一个块终止时总是执行的代码块。它与块是否正常退出、是否抛出并处理异常、是否因一个未捕获的异常而终止，这些都没关系，*ensure* 块始终都会运行。

语法

```
begin
   #.. 过程
   #.. 抛出异常
rescue
   #.. 处理错误
ensure
   #.. 最后确保执行
   #.. 这总是会执行
end
```

实例

```
begin
  raise 'A test exception.'
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
ensure
  puts "Ensuring execution"
end
```

这将产生以下结果：

```
A test exception.
["main.rb:4"]
Ensuring execution
```

使用 *else* 语句

如果提供了 *else* 子句，它一般是放置在 *rescue* 子句之后，任意 *ensure* 之前。

else 子句的主体只有在代码主体没有抛出异常时执行。

语法

```
begin
   #.. 过程
   #.. 抛出异常
rescue
   #.. 处理错误
else
   #.. 如果没有异常则执行
ensure
   #.. 最后确保执行
   #.. 这总是会执行
end
```

实例

```
begin
 # 抛出 'A test exception.'
 puts "I'm not raising exception"
rescue Exception => e
  puts e.message
  puts e.backtrace.inspect
else
   puts "Congratulations-- no errors!"
ensure
  puts "Ensuring execution"
end
```

这将产生以下结果：

```
I'm not raising exception
Congratulations-- no errors!
Ensuring execution
```

使用 $! 变量可以捕获抛出的错误消息。

Catch 和 Throw

raise 和 rescue 的异常机制能在发生错误时放弃执行，有时候需要在正常处理时跳出一些深层嵌套的结构。此时 catch 和 throw 就派上用场了。

catch 定义了一个使用给定的名称（可以是 Symbol 或 String）作为标签的块。块会正常执行知道遇到一个 throw。

语法

```
throw :lablename
#.. 这不会被执行
catch :lablename do
#.. 在遇到一个 throw 后匹配将被执行的 catch
end

OR

throw :lablename condition
#.. 这不会被执行
catch :lablename do
#.. 在遇到一个 throw 后匹配将被执行的 catch
end
```

实例

下面的实例中，如果用户键入 '!' 回应任何提示，使用一个 throw 终止与用户的交互。

```
def promptAndGet(prompt)
   print prompt
   res = readline.chomp
   throw :quitRequested if res == "!"
   return res
end

catch :quitRequested do
   name = promptAndGet("Name: ")
   age = promptAndGet("Age: ")
   sex = promptAndGet("Sex: ")
   # ..
   # 处理信息
end
promptAndGet("Name:")
```

上面的程序需要人工交互，您可以在您的计算机上进行尝试。这将产生以下结果：

```
Name: Ruby on Rails
Age: 3
Sex: !
Name:Just Ruby
```

类 Exception

Ruby 的标准类和模块抛出异常。所有的异常类组成一个层次，包括顶部的 Exception 类在内。下一层是七种不同的类型：

- Interrupt
- NoMemoryError
- SignalException
- ScriptError
- StandardError
- SystemExit

Fatal 是该层中另一种异常，但是 Ruby 解释器只在内部使用它。

ScriptError 和 StandardError 都有一些子类，但是在这里我们不需要了解这些细节。最重要的事情是创建我们自己的异常类，它们必须是类 Exception 或其子代的子类。

让我们看一个实例：

```
class FileSaveError < StandardError
   attr_reader :reason
   def initialize(reason)
      @reason = reason
   end
end
```

现在，看下面的实例，将用到上面的异常：

```
File.open(path, "w") do |file|
begin
    # 写出数据 ...
rescue
    # 发生错误
    raise FileSaveError.new($!)
end
end
```

在这里，最重要的一行是 raise *FileSaveError.new($!)*。我们调用 raise 来示意异常已经发生，把它传给 FileSaveError 的一个新的实例，由于特定的异常引起数据写入失败。

Ruby 高级

Ruby 面向对象

Ruby 是纯面向对象的语言，Ruby 中的一切都是以对象的形式出现。Ruby 中的每个值都是一个对象，即使是最原始的东西：字符串、数字，甚至连 true 和 false 都是对象。类本身也是一个对象，是 *Class* 类的一个实例。本章将向您讲解所有与 Ruby 面向对象相关的主要功能。

类用于指定对象的形式，它结合了数据表示法和方法，把数据整理成一个整齐的包。类中的数据和方法被称为类的成员。

Ruby 类定义

当您定义一个类时，您实际是定义了一个数据类型的蓝图。这实际上并没有定义任何的数据，而是定义了类的名称意味着什么，也就是说，定义了类的对象将由什么组成，以及在该对象上能执行什么操作。

类定义以关键字 **class** 开始，后跟类名称，最后以一个 **end** 进行分隔表示终止该类定义。例如，我们使用关键字 class 来定义 Box 类，如下所示：

```
class Box
   code
end
```

按照惯例，名称必须以大写字母开头，如果包含多个单词，每个单词首字母大写，但此间没有分隔符（例如：CamelCase）。

定义 Ruby 对象

类提供了对象的蓝图，所以基本上，对象是根据类进行创建的。我们使用 **new** 关键字声明类的对象。下面的语句声明了类 Box 的两个对象：

```
box1 = Box.new
box2 = Box.new
```

initialize 方法

initialize 方法是一个标准的 Ruby 类方法，与其他面向对象编程语言中的 **constructor** 工作原理类似。当您想要在创建对象的同时初始化一些类变量，initialize 方法就派上用场了。该方法带有一系列参数，与其他 Ruby 方法一样，使用该方法时，必须在前面放置 **def** 关键字，

如下所示：

```
class Box
   def initialize(w,h)
      @width, @height = w, h
   end
end
```

实例变量

实例变量是类属性，它们在使用类创建对象时就变成对象的属性。每个对象的属性是单独赋值的，和其他对象之间不共享值。在类的内部，是使用 @ 运算符访问这些属性，在类的外部，则是使用称为访问器方法的公共方法进行访问。下面我们以上面定义的类 **Box** 为实例，把 @width 和 @height 作为类 Box 的实例变量。

```
class Box
   def initialize(w,h)
      # 给实例变量赋值
      @width, @height = w, h
   end
end
```

访问器 & 设置器 方法

为了在类的外部使用变量，我们必须在访问器方法内部定义这些变量，这些访问器方法也被称为获取器方法。下面的实例演示了访问器方法的用法：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 访问器方法
   def printWidth
      @width
   end

   def printHeight
      @height
   end
end

# 创建对象
box = Box.new(10, 20)

# 使用访问器方法
x = box.printWidth()
y = box.printHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

当上面的代码执行时，它会产生以下结果：

```
Width of the box is : 10
Height of the box is : 20
```

与用于访问变量值的访问器方法类似，Ruby 提供了一种在类的外部设置变量值的方式，也就是所谓的设置器方法，定义如下：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 访问器方法
   def getWidth
      @width
   end
   def getHeight
      @height
   end

   # 设置器方法
   def setWidth=(value)
      @width = value
   end
   def setHeight=(value)
      @height = value
   end
end

# 创建对象
box = Box.new(10, 20)

# 使用设置器方法
box.setWidth = 30
box.setHeight = 50

# 使用访问器方法
x = box.getWidth()
y = box.getHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

当上面的代码执行时，它会产生以下结果：

```
Width of the box is : 30
Height of the box is : 50
```

实例方法

实例方法的定义与其他方法的定义一样，都是使用 **def** 关键字，但它们只能通过类实例来使用，如下面实例所示。它们的功能不限于访问实例变量，也能按照您的需求做更多其他的任务。

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # constructor method
   def initialize(w,h)
      @width, @height = w, h
   end
   # 实例方法
   def getArea
      @width * @height
   end
end

# 创建对象
box = Box.new(10, 20)

# 调用实例方法
a = box.getArea()
puts "Area of the box is : #{a}"
```

当上面的代码执行时，它会产生以下结果：

```
Area of the box is : 200
```

类方法 & 类变量

类变量是在类的所有实例中共享的变量。换句话说，类变量的实例可以被所有的对象实例访问。类变量以两个 @ 字符（@@）作为前缀，类变量必须在类定义中被初始化，如下面实例所示。

类方法使用 **def self.methodname()** 定义，类方法以 end 分隔符结尾。类方法可使用带有类名称的 **classname.methodname** 形式调用，如下面实例所示：

```
#!/usr/bin/ruby -w

class Box
   # 初始化类变量
   @@count = 0
   def initialize(w,h)
      # 给实例变量赋值
      @width, @height = w, h

      @@count += 1
   end

   def self.printCount()
      puts "Box count is : #@@count"
   end
end

# 创建两个对象
box1 = Box.new(10, 20)
box2 = Box.new(30, 100)

# 调用类方法来输出盒子计数
Box.printCount()
```

当上面的代码执行时，它会产生以下结果：

```
Box count is : 2
```

to_s 方法

您定义的任何类都有一个 **to_s** 实例方法来返回对象的字符串表示形式。下面是一个简单的实例，根据 width 和 height 表示 Box 对象：

```
#!/usr/bin/ruby -w

class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end
   # 定义 to_s 方法
   def to_s
      "(w:#@width,h:#@height)"  # 对象的字符串格式
   end
end

# 创建对象
box = Box.new(10, 20)

# 自动调用 to_s 方法
puts "String representation of box is : #{box}"
```

当上面的代码执行时，它会产生以下结果：

```
String representation of box is : (w:10,h:20)
```

访问控制

Ruby 为您提供了三个级别的实例方法保护，分别是 **public、private** 或 **protected**。Ruby 不在实例和类变量上应用任何访问控制。

- **Public** 方法： Public 方法可被任意对象调用。默认情况下，方法都是 public 的，除了 initialize 方法总是 private 的。
- **Private** 方法： Private 方法不能从类外部访问或查看。只有类方法可以访问私有成员。
- **Protected** 方法： Protected 方法只能被类及其子类的对象调用。访问也只能在类及其子类内部进行。

下面是一个简单的实例，演示了这三种修饰符的语法：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 实例方法默认是 public 的
   def getArea
      getWidth() * getHeight
   end

   # 定义 private 的访问器方法
   def getWidth
      @width
   end
   def getHeight
      @height
   end
   # make them private
   private :getWidth, :getHeight

   # 用于输出面积的实例方法
   def printArea
      @area = getWidth() * getHeight
      puts "Big box area is : #@area"
   end
   # 让实例方法是 protected 的
   protected :printArea
end

# 创建对象
box = Box.new(10, 20)

# 调用实例方法
a = box.getArea()
puts "Area of the box is : #{a}"

# 尝试调用 protected 的实例方法
box.printArea()
```

当上面的代码执行时，它会产生以下结果。在这里，第一种方法调用成功，但是第二方法会产生一个问题。

```
Area of the box is : 200
test.rb:42: protected method `printArea' called for #
<Box:0xb7f11280 @height=20, @width=10> (NoMethodError)
```

类的继承

继承，是面向对象编程中最重要的概念之一。继承允许我们根据另一个类定义一个类，这样使得创建和维护应用程序变得更加容易。

继承有助于重用代码和快速执行，不幸的是，Ruby 不支持多继承，但是 Ruby 支持 **mixins**。mixin 就像是多继承的一个特定实现，在多继承中，只有接口部分是可继承的。

当创建类时，程序员可以直接指定新类继承自某个已有类的成员，这样就不用从头编写新的数据成员和成员函数。这个已有类被称为基类或父类，新类被称为派生类或子类。

Ruby 也提供了子类化的概念，子类化即继承，下面的实例解释了这个概念。扩展一个类的语法非常简单。只要添加一个 < 字符和父类的名称到类语句中即可。例如，下面定义了类 *BigBox* 是 *Box* 的子类：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end
   # 实例方法
   def getArea
      @width * @height
   end
end

# 定义子类
class BigBox < Box

   # 添加一个新的实例方法
   def printArea
      @area = @width * @height
      puts "Big box area is : #@area"
   end
end

# 创建对象
box = BigBox.new(10, 20)

# 输出面积
box.printArea()
```

当上面的代码执行时，它会产生以下结果：

```
Big box area is : 200
```

方法重载

虽然您可以在派生类中添加新的功能，但有时您可能想要改变已经在父类中定义的方法的行为。这时您可以保持方法名称不变，重载方法的功能即可，如下面实例所示：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end
   # 实例方法
   def getArea
      @width * @height
   end
end

# 定义子类
class BigBox < Box

   # 改变已有的 getArea 方法
   def getArea
      @area = @width * @height
      puts "Big box area is : #@area"
   end
end

# 创建对象
box = BigBox.new(10, 20)

# 使用重载的方法输出面积
box.getArea()
```

运算符重载

我们希望使用 + 运算符执行两个 Box 对象的向量加法，使用 * 运算符来把 Box 的 width 和 height 相乘，使用一元运算符 - 对 Box 的 width 和 height 求反。下面是一个带有数学运算符定义的 Box 类版本：

```
class Box
  def initialize(w,h) # 初始化 width 和 height
    @width,@height = w, h
  end

  def +(other)         # 定义 + 来执行向量加法
    Box.new(@width + other.width, @height + other.height)
  end

  def -@               # 定义一元运算符 - 来对 width 和 height 求反
    Box.new(-@width, -@height)
  end

  def *(scalar)        # 执行标量乘法
    Box.new(@width*scalar, @height*scalar)
  end
end
```

冻结对象

有时候，我们想要防止对象被改变。在 Object 中，freeze 方法可实现这点，它能有效地把一个对象变成一个常量。任何对象都可以通过调用 **Object.freeze** 进行冻结。冻结对象不能被修改，也就是说，您不能改变它的实例变量。

您可以使用 **Object.frozen?** 方法检查一个给定的对象是否已经被冻结。如果对象已被冻结，该方法将返回 true，否则返回一个 false 值。下面的实例解释了这个概念：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 访问器方法
   def getWidth
      @width
   end
   def getHeight
      @height
   end

   # 设置器方法
   def setWidth=(value)
      @width = value
   end
   def setHeight=(value)
      @height = value
   end
end

# 创建对象
box = Box.new(10, 20)

# 让我们冻结该对象
box.freeze
if( box.frozen? )
   puts "Box object is frozen object"
else
   puts "Box object is normal object"
end

# 现在尝试使用设置器方法
box.setWidth = 30
box.setHeight = 50

# 使用访问器方法
x = box.getWidth()
y = box.getHeight()

puts "Width of the box is : #{x}"
puts "Height of the box is : #{y}"
```

当上面的代码执行时，它会产生以下结果：

```
Box object is frozen object
test.rb:20:in `setWidth=': can't modify frozen object (TypeError)
        from test.rb:39
```

类常量

您可以在类的内部定义一个常量，通过把一个直接的数值或字符串值赋给一个变量来定义的，常量的定义不需要使用 @ 或 @@。按照惯例，常量的名称使用大写。

一旦常量被定义，您就不能改变它的值，您可以在类的内部直接访问常量，就像是访问变量一样，但是如果您想要在类的外部访问常量，那么您必须使用 **classname::constant**，如下面实例所示。

```
#!/usr/bin/ruby -w

# 定义类
class Box
   BOX_COMPANY = "TATA Inc"
   BOXWEIGHT = 10
   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end
   # 实例方法
   def getArea
      @width * @height
   end
end

# 创建对象
box = Box.new(10, 20)

# 调用实例方法
a = box.getArea()
puts "Area of the box is : #{a}"
puts Box::BOX_COMPANY
puts "Box weight is: #{Box::BOXWEIGHT}"
```

当上面的代码执行时，它会产生以下结果：

```
Area of the box is : 200
TATA Inc
Box weight is: 10
```

类常量可被继承，也可像实例方法一样被重载。

使用 allocate 创建对象

可能有一种情况，您想要在不调用对象构造器 **initialize** 的情况下创建对象，即，使用 new 方法创建对象，在这种情况下，您可以调用 allocate 来创建一个未初始化的对象，如下面实例所示：

```
#!/usr/bin/ruby -w

# 定义类
class Box
   attr_accessor :width, :height

   # 构造器方法
   def initialize(w,h)
      @width, @height = w, h
   end

   # 实例方法
   def getArea
      @width * @height
   end
end

# 使用 new 创建对象
box1 = Box.new(10, 20)

# 使用 allocate 创建两一个对象
box2 = Box.allocate

# 使用 box1 调用实例方法
a = box1.getArea()
puts "Area of the box is : #{a}"

# 使用 box2 调用实例方法
a = box2.getArea()
puts "Area of the box is : #{a}"
```

当上面的代码执行时，它会产生以下结果：

```
Area of the box is : 200
test.rb:14: warning: instance variable @width not initialized
test.rb:14: warning: instance variable @height not initialized
test.rb:14:in `getArea': undefined method `*'
   for nil:NilClass (NoMethodError) from test.rb:29
```

类信息

如果类定义是可执行代码，这意味着，它们可在某个对象的上下文中执行，self 必须引用一些东西。让我们来看看下面的实例：.

```
#!/usr/bin/ruby -w

class Box
   # 输出类信息
   puts "Type of self = #{self.type}"
   puts "Name of self = #{self.name}"
end
```

当上面的代码执行时，它会产生以下结果：

```
Type of self = Class
Name of self = Box
```

这意味着类定义可通过把该类作为当前对象来执行，同时也意味着元类和父类中的该方法在方法定义执行期间是可用的。

Ruby 正则表达式

正则表达式是一种特殊序列的字符，它通过使用有专门语法的模式来匹配或查找其他字符串或字符串集合。

语法

正则表达式从字面上看是一种介于斜杠之间或介于跟在 %r 后的任意分隔符之间的模式，如下所示：

```
/pattern/
/pattern/im    # 可以指定选项
%r!/usr/local! # 一般的分隔的正则表达式
```

实例

```
#!/usr/bin/ruby

line1 = "Cats are smarter than dogs";
line2 = "Dogs also like meat";

if ( line1 =~ /Cats(.*)/ )
  puts "Line1 contains Cats"
end
if ( line2 =~ /Cats(.*)/ )
  puts "Line2 contains  Dogs"
end
```

这将产生以下结果：

```
Line1 contains Cats
```

正则表达式修饰符

正则表达式从字面上看可能包含一个可选的修饰符，用于控制各方面的匹配。修饰符在第二个斜杠字符后指定，如上面实例所示。下标列出了 可能的修饰符：

修饰符	描述
i	当匹配文本时忽略大小写。
o	只执行一次 #{} 插值，正则表达式在第一次时就进行判断。
x	忽略空格，允许在正则表达式中进行注释。
m	匹配多行，把换行字符识别为正常字符。
u,e,s,n	把正则表达式解释为 Unicode（UTF-8）、EUC、SJIS 或 ASCII。如果没有指定修饰符，则认为正则表达式使用的是源编码。

就像字符串通过 %Q 进行分隔一样，Ruby 允许您以 %r 作为正则表达式的开头，后面跟着任意分隔符。这在描述包含大量您不想转义的斜杠字符时非常有用。

```
# 下面匹配单个斜杠字符，不转义
%r|/|

# Flag 字符可通过下面的语法进行匹配
%r[</(.*)>]i
```

正则表达式模式

除了控制字符，**(+ ? . * ^ $ ( ) [ ] { } |)**，其他所有字符都匹配本身。您可以通过在控制字符前放置一个反斜杠来对控制字符进行转义。

下表列出了 Ruby 中可用的正则表达式语法。

模式	描述
^	匹配行的开头。
$	匹配行的结尾。
.	匹配除了换行符以外的任意单字符。使用 m 选项时，它也可以匹配换行符。
[...]	匹配在方括号中的任意单字符。
[^...]	匹配不在方括号中的任意单字符。
re*	匹配前面的子表达式零次或多次。
re+	匹配前面的子表达式一次或多次。
re?	匹配前面的子表达式零次或一次。
re{ n}	匹配前面的子表达式 n 次。
re{ n,}	匹配前面的子表达式 n 次或 n 次以上。
re{ n, m}	匹配前面的子表达式至少 n 次至多 m 次。
a| b	匹配 a 或 b。

(re)	对正则表达式进行分组，并记住匹配文本。
(?imx)	暂时打开正则表达式内的 i、 m 或 x 选项。如果在圆括号中，则只影响圆括号内的部分。
(?-imx)	暂时关闭正则表达式内的 i、 m 或 x 选项。如果在圆括号中，则只影响圆括号内的部分。
(?: re)	对正则表达式进行分组，但不记住匹配文本。
(?imx: re)	暂时打开圆括号内的 i、 m 或 x 选项。
(?-imx: re)	暂时关闭圆括号内的 i、 m 或 x 选项。
(?#...)	注释。
(?= re)	使用模式指定位置。没有范围。
(?! re)	使用模式的否定指定位置。没有范围。
(?> re)	匹配无回溯的独立模式。
\w	匹配单词字符。
\W	匹配非单词字符。
\s	匹配空白字符。等价于 [\t\n\r\f]。
\S	匹配非空白字符。
\d	匹配数字。等价于 [0-9]。
\D	匹配非数字。
\A	匹配字符串的开头。
\Z	匹配字符串的结尾。如果存在换行符，则只匹配到换行符之前。
\z	匹配字符串的结尾。
\G	匹配最后一个匹配完成的点。
\b	当在括号外时匹配单词边界，当在括号内时匹配退格键（0x08）。
\B	匹配非单词边界。
\n, \t, etc.	匹配换行符、回车符、制表符，等等。
\1...\9	匹配第 n 个分组子表达式。
\10	如果已匹配过，则匹配第 n 个分组子表达式。否则指向字符编码的八进制表示。

正则表达式实例

字符

实例	描述
/ruby/	匹配 "ruby"
¥	匹配 Yen 符号。Ruby 1.9 和 Ruby 1.8 支持多个字符。

字符类

实例	描述
/[Rr]uby/	匹配 "Ruby" 或 "ruby"
/rub[ye]/	匹配 "ruby" 或 "rube"
/[aeiou]/	匹配任何一个小写元音字母
/[0-9]/	匹配任何一个数字，与 /[0123456789]/ 相同
/[a-z]/	匹配任何一个小写 ASCII 字母
/[A-Z]/	匹配任何一个大写 ASCII 字母
/[a-zA-Z0-9]/	匹配任何一个括号内的字符
/[^aeiou]/	匹配任何一个非小写元音字母的字符
/[^0-9]/	匹配任何一个非数字字符

特殊字符类

实例	描述
/./	匹配除了换行符以外的其他任意字符
/./m	在多行模式下，也能匹配换行符
/\d/	匹配一个数字，等同于 /[0-9]/
/\D/	匹配一个非数字，等同于 /[^0-9]/
/\s/	匹配一个空白字符，等同于 /[\t\r\n\f]/
/\S/	匹配一个非空白字符，等同于 /[^ \t\r\n\f]/
/\w/	匹配一个单词字符，等同于 /[A-Za-z0-9_]/
/\W/	匹配一个非单词字符，等同于 /[^A-Za-z0-9_]/

重复

实例	描述
/ruby?/	匹配 "rub" 或 "ruby"。其中，y 是可有可无的。
/ruby*/	匹配 "rub" 加上 0 个或多个的 y。
/ruby+/	匹配 "rub" 加上 1 个或多个的 y。
/\d{3}/	刚好匹配 3 个数字。
/\d{3,}/	匹配 3 个或多个数字。
/\d{3,5}/	匹配 3 个、4 个或 5 个数字。

非贪婪重复

这会匹配最小次数的重复。

实例	描述
`/<.*>/`	贪婪重复：匹配 " `<ruby>perl>` "
`/<.*?>/`	非贪婪重复：匹配 " `<ruby>perl>` " 中的 " `<ruby>` "

通过圆括号进行分组

实例	描述
/\D\d+/	无分组： + 重复 \d
/(\D\d)+/	分组： + 重复 \D\d 对
/([Rr]uby(,)?)+/	匹配 "Ruby"、"Ruby, ruby, ruby"，等等

反向引用

这会再次匹配之前匹配过的分组。

实例	描述
/([Rr])uby&\1ails/	匹配 ruby&rails 或 Ruby&Rails
/(['"])(?:(?!\1).)*\1/	单引号或双引号字符串。\1 匹配第一个分组所匹配的字符，\2 匹配第二个分组所匹配的字符，依此类推。

替换

实例	描述
/ruby|rube/	匹配 "ruby" 或 "rube"
/rub(y|le))/	匹配 "ruby" 或 "ruble"
/ruby(!+|\?)/	"ruby" 后跟一个或多个 ! 或者跟一个 ?

锚

这需要指定匹配位置。

实例	描述
/^Ruby/	匹配以 "Ruby" 开头的字符串或行
/Ruby$/	匹配以 "Ruby" 结尾的字符串或行
/\ARuby/	匹配以 "Ruby" 开头的字符串
/Ruby\Z/	匹配以 "Ruby" 结尾的字符串
/\bRuby\b/	匹配单词边界的 "Ruby"
/\brub\B/	\B 是非单词边界：匹配 "rube" 和 "ruby" 中的 "rub"，但不匹配单独的 "rub"
/Ruby(?=!)/	如果 "Ruby" 后跟着一个感叹号，则匹配 "Ruby"
/Ruby(?!!)/	如果 "Ruby" 后没有跟着一个感叹号，则匹配 "Ruby"

圆括号的特殊语法

实例	描述
/R(?#comment)/	匹配 "R"。所有剩余的字符都是注释。
/R(?i)uby/	当匹配 "uby" 时不区分大小写。
/R(?i:uby)/	与上面相同。
/rub(?:y|le))/	只分组，不进行 \1 反向引用

搜索和替换

sub 和 **gsub** 及它们的替代变量 **sub!** 和 **gsub!** 是使用正则表达式时重要的字符串方法。

所有这些方法都是使用正则表达式模式执行搜索与替换操作。**sub** 和 **sub!** 替换模式的第一次出现，**gsub** 和 **gsub!** 替换模式的所有出现。

sub 和 **gsub** 返回一个新的字符串，保持原始的字符串不被修改，而 **sub!** 和 **gsub!** 则会修改它们调用的字符串。

下面是一个实例：

```
#!/usr/bin/ruby

phone = "2004-959-559 #This is Phone Number"

# 删除 Ruby 的注释
phone = phone.sub!(/#.*$/, "")
puts "Phone Num : #{phone}"

# 移除数字以外的其他字符
phone = phone.gsub!(/\D/, "")
puts "Phone Num : #{phone}"
```

这将产生以下结果：

```
Phone Num : 2004-959-559
Phone Num : 2004959559
```

下面是另一个实例：

```
#!/usr/bin/ruby

text = "rails are rails, really good Ruby on Rails"

# 把所有的 "rails" 改为 "Rails"
text.gsub!("rails", "Rails")

# 把所有的单词 "Rails" 都改成首字母大写
text.gsub!(/\brails\b/, "Rails")

puts "#{text}"
```

这将产生以下结果：

```
Rails are Rails, really good Ruby on Rails
```

Ruby 数据库访问 - DBI 教程

本章节将向您讲解如何使用 Ruby 访问数据库。*Ruby DBI* 模块为 Ruby 脚本提供了类似于 Perl DBI 模块的独立于数据库的接口。

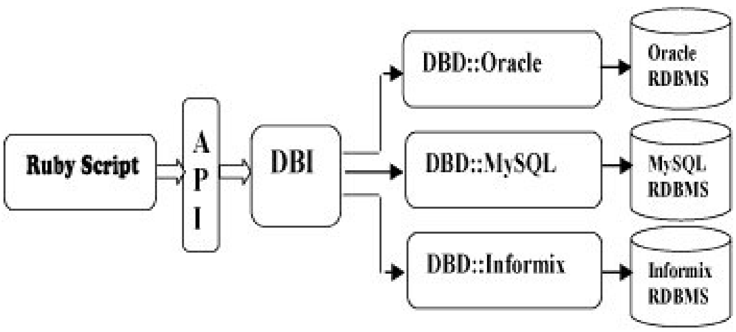
DBI 即 Database independent interface，代表了 Ruby 独立于数据库的接口。DBI 在 Ruby 代码与底层数据库之间提供了一个抽象层，允许您简单地实现数据库切换。它定义了一系列方法、变量和规范，提供了一个独立于数据库的一致的数据库接口。

DBI 可与下列进行交互：

- ADO (ActiveX Data Objects)
- DB2
- Frontbase
- mSQL
- MySQL
- ODBC
- Oracle
- OCI8 (Oracle)
- PostgreSQL
- Proxy/Server
- SQLite
- SQLRelay

DBI 应用架构

DBI 独立于任何在后台中可用的数据库。无论您使用的是 Oracle、MySQL、Informix，您都可以使用 DBI。下面的架构图清晰地说明了这点。



Ruby DBI 一般的架构使用两个层：

- 数据库接口（DBI）层。该层是独立于数据库，并提供了一系列公共访问方法，方法的使用不分数据库服务器类型。

- 数据库驱动（DBD）层。该层是依赖于数据库，不同的驱动提供了对不同的数据库引擎的访问。MySQL、PostgreSQL、InterBase、Oracle 等分别使用不同的驱动。每个驱动都负责解释来自 DBI 层的请求，并把这些请求映射为适用于给定类型的数据库服务器的请求。

先决条件

如果您想要编写 Ruby 脚本来访问 MySQL 数据库，您需要先安装 Ruby MySQL 模块。

该模块是一个 DBD，可从 http://www.tmtm.org/en/mysql/ruby/ 上下载。

获取并安装 Ruby/DBI

您可以从下面的链接下载并安装 Ruby DBI 模块：

http://rubyforge.org/projects/ruby-dbi/

在开始安装之前，请确保您拥有 root 权限。现在，请安装下面的步骤进行安装：

步骤 1

```
$ tar zxf dbi-0.2.0.tar.gz
```

步骤 2

进入目录 *dbi-0.2.0*，在目录中使用 *setup.rb* 脚本进行配置。最常用的配置命令是 config 参数后不跟任何参数。该命令默认配置为安装所有的驱动。

```
$ ruby setup.rb config
```

更具体地，您可以使用 --with 选项来列出了您要使用的特定部分。例如，如果只想配置主要的 DBI 模块和 MySQL DBD 层驱动，请输入下面的命令：

```
$ ruby setup.rb config --with=dbi,dbd_mysql
```

步骤 3

最后一步是建立驱动器，使用下面命令进行安装：

```
$ ruby setup.rb setup
$ ruby setup.rb install
```

数据库连接

假设我们使用的是 MySQL 数据库，在连接数据库之前，请确保：

- 您已经创建了一个数据库 TESTDB。
- 您已经在 TESTDB 中创建了表 EMPLOYEE。
- 该表带有字段 FIRST_NAME、LAST_NAME、AGE、SEX 和 INCOME。
- 设置用户 ID "testuser" 和密码 "test123" 来访问 TESTDB
- 已经在您的机器上正确地安装了 Ruby 模块 DBI。
- 您已经看过 MySQL 教程，理解了 MySQL 基础操作。

下面是连接 MySQL 数据库 "TESTDB" 的实例：

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     # 获取服务器版本字符串，并显示
     row = dbh.select_one("SELECT VERSION()")
     puts "Server version: " + row[0]
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

当运行这段脚本时，将会在 Linux 机器上产生以下结果。

```
Server version: 5.0.45
```

如果建立连接时带有数据源，则返回数据库句柄（Database Handle），并保存到 **dbh** 中以便后续使用，否则 **|dbh** 将被设置为 nil 值，*e.err* 和 *e::errstr* 分别返回错误代码和错误字符串。

最后，在退出这段程序之前，请确保关闭数据库连接，释放资源。

INSERT 操作

当您想要在数据库表中创建记录时，需要用到 INSERT 操作。

一旦建立了数据库连接，我们就可以准备使用 **do** 方法或 **prepare** 和 **execute** 方法创建表或创建插入数据表中的记录。

使用 do 语句

不返回行的语句可通过调用 **do** 数据库处理方法。该方法带有一个语句字符串参数，并返回该语句所影响的行数。

```
dbh.do("DROP TABLE IF EXISTS EMPLOYEE")
dbh.do("CREATE TABLE EMPLOYEE (
     FIRST_NAME  CHAR(20) NOT NULL,
     LAST_NAME  CHAR(20),
     AGE INT,
     SEX CHAR(1),
     INCOME FLOAT )" );
```

同样地，您可以执行 SQL *INSERT* 语句来创建记录插入 EMPLOYEE 表中。

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     dbh.do( "INSERT INTO EMPLOYEE(FIRST_NAME,
                   LAST_NAME,
                   AGE,
           SEX,
           INCOME)
          VALUES ('Mac', 'Mohan', 20, 'M', 2000)" )
     puts "Record has been created"
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

使用 *prepare* 和 *execute*

您可以使用 DBI 的 *prepare* 和 *execute* 方法来执行 Ruby 代码中的 SQL 语句。

创建记录的步骤如下：

- 准备带有 INSERT 语句的 SQL 语句。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** API 来完成。

- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

下面是使用这两种方法的语法：

```
sth = dbh.prepare(statement)
sth.execute
   ... zero or more SQL operations ...
sth.finish
```

这两种方法可用于传 **bind** 值给 SQL 语句。有时候被输入的值可能未事先给出，在这种情况下，则会用到绑定值。使用问号（**?**）替代实际值，实际值通过 execute() API 来传递。

下面的实例在 EMPLOYEE 表中创建了两个记录：

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare( "INSERT INTO EMPLOYEE(FIRST_NAME,
                   LAST_NAME,
                   AGE,
            SEX,
           INCOME)
                   VALUES (?, ?, ?, ?, ?)" )
     sth.execute('John', 'Poul', 25, 'M', 2300)
     sth.execute('Zara', 'Ali', 17, 'F', 1000)
     sth.finish
     dbh.commit
     puts "Record has been created"
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

如果同时使用多个 INSERT，那么先准备一个语句，然后在一个循环中多次执行它要比通过循环每次调用 do 有效率得多。

READ 操作

对任何数据库的 READ 操作是指从数据库中获取有用的信息。

一旦建立了数据库连接，我们就可以准备查询数据库。我们可以使用 **do** 方法或 **prepare** 和 **execute** 方法从数据库表中获取值。

获取记录的步骤如下：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。
- 逐一获取结果，并输出这些结果。这将通过使用 **fetch** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** 方法来完成。

下面的实例从 EMPLOYEE 表中查询所有工资（salary）超过 1000 的记录。

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare("SELECT * FROM EMPLOYEE
                       WHERE INCOME > ?")
     sth.execute(1000)

     sth.fetch do |row|
        printf "First Name: %s, Last Name : %s\n", row[0], row[1]
        printf "Age: %d, Sex : %s\n", row[2], row[3]
        printf "Salary :%d \n\n", row[4]
     end
     sth.finish
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

这将产生以下结果：

```
First Name: Mac, Last Name : Mohan
Age: 20, Sex : M
Salary :2000

First Name: John, Last Name : Poul
Age: 25, Sex : M
Salary :2300
```

还有很多从数据库获取记录的方法，如果您感兴趣，可以查看 Ruby DBI Read 操作。

Update 操作

对任何数据库的 UPDATE 操作是指更新数据库中一个或多个已有的记录。下面的实例更新 SEX 为 'M' 的所有记录。在这里，我们将把所有男性的 AGE 增加一岁。这将分为三步：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中选择所有的结果。这将通过使用 **execute** 方法来完成。

- 释放语句句柄。这将通过使用 **finish** 方法来完成。
- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare("UPDATE EMPLOYEE SET AGE = AGE + 1
                       WHERE SEX = ?")
     sth.execute('M')
     sth.finish
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

DELETE 操作

当您想要从数据库中删除记录时，需要用到 DELETE 操作。下面的实例从 EMPLOYEE 中删除 AGE 超过 20 的所有记录。该操作的步骤如下：

- 基于所需的条件准备 SQL 查询。这将通过使用 **prepare** 方法来完成。
- 执行 SQL 查询，从数据库中删除所需的记录。这将通过使用 **execute** 方法来完成。
- 释放语句句柄。这将通过使用 **finish** 方法来完成。
- 如果一切进展顺利，则 **commit** 该操作，否则您可以 **rollback** 完成交易。

```
#!/usr/bin/ruby -w

require "dbi"

begin
     # 连接到 MySQL 服务器
     dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                       "testuser", "test123")
     sth = dbh.prepare("DELETE FROM EMPLOYEE
                       WHERE AGE > ?")
     sth.execute(20)
     sth.finish
     dbh.commit
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

执行事务

事务是一种确保交易一致性的机制。事务应具有下列四种属性：

- 原子性（**Atomicity**）：事务的原子性指的是，事务中包含的程序作为数据库的逻辑工作单位，它所做的对数据修改操作要么全部执行，要么完全不执行。
- 一致性（**Consistency**）：事务的一致性指的是在一个事务执行之前和执行之后数据库都必须处于一致性状态。假如数据库的状态满足所有的完整性约束，就说该数据库是一致的。
- 隔离性（**Isolation**）：事务的隔离性指并发的事务是相互隔离的，即一个事务内部的操作及正在操作的数据必须封锁起来，不被其它企图进行修改的事务看到。
- 持久性（**Durability**）：事务的持久性意味着当系统或介质发生故障时，确保已提交事务的更新不能丢失。即一旦一个事务提交，它对数据库中数据的改变应该是永久性的，耐得住任何数据库系统故障。持久性通过数据库备份和恢复来保证。

DBI 提供了两种执行事务的方法。一种是 *commit* 或 *rollback* 方法，用于提交或回滚事务。还有一种是 *transaction* 方法，可用于实现事务。接下来我们来介绍这两种简单的实现事务的方法：

方法 I

第一种方法使用 DBI 的 *commit* 和 *rollback* 方法来显式地提交或取消事务：

```
dbh['AutoCommit'] = false # 设置自动提交为 false.
begin
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'John'")
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'Zara'")
  dbh.commit
rescue
  puts "transaction failed"
  dbh.rollback
end
dbh['AutoCommit'] = true
```

方法 II

第二种方法使用 *transaction* 方法。这个方法相对简单些，因为它需要一个包含构成事务语句的代码块。*transaction* 方法执行块，然后根据块是否执行成功，自动调用 *commit* 或 *rollback*：

```
dbh['AutoCommit'] = false # 设置自动提交为 false
dbh.transaction do |dbh|
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'John'")
  dbh.do("UPDATE EMPLOYEE SET AGE = AGE+1
          WHERE FIRST_NAME = 'Zara'")
end
dbh['AutoCommit'] = true
```

COMMIT 操作

Commit 是一种标识数据库已完成更改的操作，在这个操作后，所有的更改都不可恢复。

下面是一个调用 **commit** 方法的简单实例。

```
dbh.commit
```

ROLLBACK 操作

如果您不满意某个或某几个更改，您想要完全恢复这些更改，则使用 **rollback** 方法。

下面是一个调用 **rollback** 方法的简单实例。

```
dbh.rollback
```

断开数据库

如需断开数据库连接，请使用 disconnect API。

```
dbh.disconnect
```

如果用户通过 disconnect 方法关闭了数据库连接，DBI 会回滚所有未完成的事务。但是，不需要依赖于任何 DBI 的实现细节，您的应用程序就能很好地显式调用 commit 或 rollback。

处理错误

有许多不同的错误来源。比如在执行 SQL 语句时的语法错误，或者是连接失败，又或者是对一个已经取消的或完成的语句句柄调用 fetch 方法。

如果某个 DBI 方法失败，DBI 会抛出异常。DBI 方法会抛出任何类型的异常，但是最重要的两种异常类是 *DBI::InterfaceError* 和 *DBI::DatabaseError*。

这些类的 Exception 对象有 *err*、*errstr* 和 *state* 三种属性，分表代表了错误号、一个描述性的错误字符串和一个标准的错误代码。属性具体说明如下：

- **err**：返回所发生的错误的整数表示法，如果 DBD 不支持则返回 *nil*。例如，Oracle DBD 返回 *ORA-XXXX* 错误消息的数字部分。
- **errstr**：返回所发生的错误的字符串表示法。
- **state**：返回所发生的错误的 SQLSTATE 代码。SQLSTATE 是五字符长度的字符串。大多数的 DBD 并不支持它，所以会返回 nil。

在上面的实例中您已经看过下面的代码：

```
rescue DBI::DatabaseError => e
     puts "An error occurred"
     puts "Error code:    #{e.err}"
     puts "Error message: #{e.errstr}"
     dbh.rollback
ensure
     # 断开与服务器的连接
     dbh.disconnect if dbh
end
```

为了获取脚本执行时有关脚本执行内容的调试信息，您可以启用跟踪。为此，您必须首先下载 dbi/trace 模块，然后调用控制跟踪模式和输出目的地的 *trace* 方法：

```
require "dbi/trace"
..............

trace(mode, destination)
```

mode 的值可以是 0（off）、1、2 或 3，destination 的值应该是一个 IO 对象。默认值分别是 2 和 STDERR。

方法的代码块

有一些创建句柄的方法。这些方法通过代码块调用。使用带有方法的代码块的优点是，它们为代码块提供了句柄作为参数，当块终止时会自动清除句柄。下面是一些实例，有助于理解这个概念。

- **DBI.connect** ：该方法生成一个数据库句柄，建议在块的末尾调用 *disconnect* 来断开数据库。
- **dbh.prepare** ：|该方法生成一个语句句柄，建议在块的末尾调用 *finish*。在块内，您必须调用 *execute* 方法来执行语句。
- **dbh.execute** ：|该方法与 dbh.prepare 类似，但是 dbh.execute 不需要在块内调用 execute 方法。语句句柄会自动执行。

实例 1

DBI.connect 可带有一个代码块，向它传递数据库句柄，且会在块的末尾自动断开句柄。

```
dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                  "testuser", "test123") do |dbh|
```

实例 2

dbh.prepare | 可带有一个代码块，向它传递语句句柄，且会在块的末尾自动调用 finish。

```
dbh.prepare("SHOW DATABASES") do |sth|
       sth.execute
       puts "Databases: " + sth.fetch_all.join(", ")
end
```

实例 3

dbh.execute | 可带有一个代码块，向它传递语句句柄，且会在块的末尾自动调用 finish。

```
dbh.execute("SHOW DATABASES") do |sth|
   puts "Databases: " + sth.fetch_all.join(", ")
end
```

DBI *transaction* 方法也可带有一个代码块，这在上面的章节中已经讲解过了。

特定驱动程序的函数和属性

DBI 让数据库驱动程序提供了额外的特定数据库的函数，这些函数可被用户通过任何 Handle 对象的 *func* 方法进行调用。

使用 **[]=** or **[]** 方法可以设置或获取特定驱动程序的属性。

DBD::Mysql 实现了下列特定驱动程序的函数：

函数	描述
dbh.func(:createdb, db_name)	创建一个新的数据库。
dbh.func(:dropdb, db_name)	删除一个数据库。
dbh.func(:reload)	执行重新加载操作。
dbh.func(:shutdown)	关闭服务器。
dbh.func(:insert_id) => Fixnum	返回该连接的最近 AUTO_INCREMENT 值。
dbh.func(:client_info) => String	根据版本返回 MySQL 客户端信息。
dbh.func(:client_version) => Fixnum	根据版本返回客户端信息。这与 :client_info 类似，但是它会返回一个 fixnum，而不是返回字符串。
dbh.func(:host_info) => String	返回主机信息。
dbh.func(:proto_info) => Fixnum	返回用于通信的协议。
dbh.func(:server_info) => String	根据版本返回 MySQL 服务器端信息。
dbh.func(:stat) => Stringb> 返回数据库的当前状态。	
dbh.func(:thread_id) => Fixnum	返回当前线程的 ID。

实例

```
#!/usr/bin/ruby

require "dbi"
begin
   # 连接到 MySQL 服务器
   dbh = DBI.connect("DBI:Mysql:TESTDB:localhost",
                     "testuser", "test123")
   puts dbh.func(:client_info)
   puts dbh.func(:client_version)
   puts dbh.func(:host_info)
   puts dbh.func(:proto_info)
   puts dbh.func(:server_info)
   puts dbh.func(:thread_id)
   puts dbh.func(:stat)
rescue DBI::DatabaseError => e
   puts "An error occurred"
   puts "Error code:    #{e.err}"
   puts "Error message: #{e.errstr}"
ensure
   dbh.disconnect if dbh
end
```

这将产生以下结果：

```
5.0.45
50045
Localhost via UNIX socket
10
5.0.45
150621
Uptime: 384981  Threads: 1  Questions: 1101078  Slow queries: 4 \
Opens: 324  Flush tables: 1  Open tables: 64  \
Queries per second avg: 2.860
```

Ruby CGI 编程

Ruby 是一门通用的语言，不仅仅是一门应用于WEB开发的语言，但 Ruby 在WEB应用及WEB工具中的开发是最常见的。

使用Ruby您不仅可以编写自己的SMTP服务器，FTP程序，或Ruby Web服务器，而且还可以使用Ruby进行CGI编程。

接下来，让我们花点时间来学校Ruby的CGI编辑。

编写 CGI 脚本

最脚本的 Ruby CGI 代码如下所示：

```
#!/usr/bin/ruby puts "HTTP/1.0 200 OK" puts "Content-type: text/html\n\n" puts "This is a
```

你可以将该代码保持到 test.cgi 文件中，上次到服务器并赋予足够权限，即可作为 CGI 脚本执行。

如果你站的的地址为http://www.example.com/ ，即可用过http://www.example.com/test.cgi 访问该程序，输出结果为： "This is a test."。

浏览器访问该网址后，Web 服务器会在站点目录下找到 test.cgi文件，然后通过Ruby解析器来解析脚本代码并访问HTML文档。

使用 cgi.rb

Ruby 可以调用 CGI 库来编写更复杂的CGI脚本。

以下代码调用了 CGI 库来创建一个脚本的CGI脚本。

```
#!/usr/bin/ruby

require 'cgi'

cgi = CGI.new
puts cgi.header
puts "<html><body>This is a test</body></html>"
```

以下代码中，创建了CGI 对象并打印头部信息。

表单处理

使用CGI库可以通过两种方式获取表单提交（或URL中的参数）的数据， 例如URL：/cgi-bin/test.cgi?FirstName=Zara&LastName=Ali。

你可以使用 CGI#[] 来直接获取参数FirstName和LastName：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi['FirstName'] # =>  ["Zara"]
cgi['LastName']  # =>  ["Ali"]
```

另外一种获取表单数据的方法：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
h = cgi.params  # =>  {"FirstName"=>["Zara"],"LastName"=>["Ali"]}
h['FirstName']  # =>  ["Zara"]
h['LastName']   # =>  ["Ali"]
```

以下代码用于检索所有的键值：

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi.keys         # =>  ["FirstName", "LastName"]
```

如果表单包含了多个相同名称的字段，则该相同字段的值将保存在数组中。

以下实例中，指定表单中三个相同的字段"name"， 值分别为 "Zara", "Huma" 和 "Nuha":

```
#!/usr/bin/ruby

require 'cgi'
cgi = CGI.new
cgi['name']        # => "Zara"
cgi.params['name'] # => ["Zara", "Huma", "Nuha"]
cgi.keys           # => ["name"]
cgi.params         # => {"name"=>["Zara", "Huma", "Nuha"]}
```

注意：Ruby 会自动判断 GET 和 POST 方法，所以无需对两种方法区别对待。

以下是相关的HML代码：

```
<html>
<body>
<form method="POST" action="http://www.example.com/test.cgi">
First Name :<input type="text" name="FirstName" value="" />
<br />
Last Name :<input type="text" name="LastName" value="" />

<input type="submit" value="Submit Data" />
</form>
</body>
</html>
```

创建 Form 表单和 HTML

CGI 包含了大量的方法来创建 HTML，每个HTML标签都有相对应的方法。 在使用这些方法前，比必须通过 CGI.new 来创建 CGI 对象。

为了使标签的嵌套更加的简单，这些方法将内容作为了代码块，代码块将返回字符串作为标签的内容。如下所示：

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cgi.out{
   cgi.html{
      cgi.head{ "\n"+cgi.title{"This Is a Test"} } +
      cgi.body{ "\n"+
         cgi.form{"\n"+
            cgi.hr +
            cgi.h1 { "A Form: " } + "\n"+
            cgi.textarea("get_text") +"\n"+
            cgi.br +
            cgi.submit
         }
      }
   }
}
```

字符串转义

当你在处理 URL 中的参数或者 HTML 表单数据时，需要对指定的特殊字符进行转义，如：引号（"）,反斜杠(/)。

Ruby CGI 对象提供了CGI.escape 和 CGI.unescape 方法来处理这些特殊字符的转义：

```
#!/usr/bin/ruby

require 'cgi'
puts CGI.escape(Zara Ali/A Sweet & Sour Girl")
```

以上代码执行结果如下：

```
#!/usr/bin/ruby

require 'cgi'
puts CGI.escape(Zara Ali/A Sweet & Sour Girl")
```

另一组实例：

```
#!/usr/bin/ruby

require 'cgi'
puts CGI.escapeHTML('<h1>Zara Ali/A Sweet & Sour Girl</h1>')
```

以上代码执行结果如下：

```
&lt;h1&gt;Zara Ali/A Sweet & Sour Girl&lt;/h1&gt;'
```

CGI 类中常用的方法

以下是Ruby中完整的CGI类的相关方法

- Ruby CGI - 标准 CGI 库相关方法

Cookies 和 Sessions

- Ruby CGI Cookies - 如何处理 CGI Cookies.
- Ruby CGI Sessions - 如何处理 CGI sessions.

Ruby CGI方法

以下为CGI类的方法列表：

序号	方法描述
CGI::new([level="query"])	创建 CGI 对象。query可以是以下值: **query:** 没有 HTML 生成输出 **html3:** HTML3.2 **html4:** HTML4.0 Strict **html4Tr:** HTML4.0 Transitional **html4Fr:** HTML4.0 Frameset
CGI::escape(str)	使用 URL 编码来转义字符串
CGI::unescape(str)	对通过 escape() 编码的字符串进行解码。
CGI::escapeHTML(str)	编码 HTML 特殊字符, 包括: & < >。
CGI::unescapeHTML(str)	解码 HTML 特殊字符, 包括: & < >。
CGI::escapeElement(str[, element...])	在指定的 HTML 元素中编码 HTML 特殊字符。
CGI::unescapeElement(str, element[, element...])	在指定的 HTML 元素中解码 HTML 特殊字符。
CGI::parse(query)	解析查询字符串，并返回包含哈希的 键=》值 对。
CGI::pretty(string[, leader=" "])	返回整齐的HTML格式。 如果指定了 *leader*， 它将写入到每一行的开头。 *leader* 默认值为两个空格。
CGI::rfc1123_date(time)	根据 RFC-1123 来格式化时间 (例如, Tue, 2 Jun 2008 00:00:00 GMT)。

CGI 实例化方法

以下实例中我们将 CGI::new 的对象赋值给 c 变量，方法列表如下：

序号	方法描述
c[name]	返回一个数组，包含了对应字段名为 *name* 的值。
c.checkbox(name[, value[, check=false]]) c.checkbox(options)	返回 HTML 字符串用于定义 checkbox 字段。标签的属性可以以一个哈希函数作为参数传递。
c.checkbox_group(name, value...) c.checkbox_group(options)	>返回 HTML 字符串用于定义 checkbox 组。标签的属性可以以一个哈希函数作为参数传递。
c.file_field(name[, size=20[,	返回定义 file 字段的HTML字符串。

max]]) c.file_field(options)	返回定义 file 字段的HTML字符串。
c.form([method="post"[, url]]) { ...} c.form(options)	返回定义 form 表单的HTML字符串。 如果指定了代码块，将作为表单内容输出。标签的属性可以以一个哈希函数作为参数传递。
c.cookies	返回 CGI::Cookie 对象，包含了cookie 中的键值对。
c.header([header])	返回 CGI 头部的信息。如果 header 参数是哈希值，其键 - 值对，用于创建头部信息。
c.hidden(name[, value]) c.hidden(options)	返回定义一个隐藏字段的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.image_button(url[, name[, alt]]) c.image_button(options)	返回定义一个图像按钮的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.keys	返回一个数组，包含了表单的字段名。
c.key?(name) c.has_key?(name) c.include?(name)	如果表单包含了指定的字段名返回 true。
c.multipart_form([url[, encode]]) { ...} c.multipart_form(options) { ...}	返回定义一个多媒体表单（multipart）的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.out([header]) { ...}	生成 HTML 并输出。使用由块的输出来创建页面的主体生成的字符串。
c.params	返回包含表单字段名称和值的哈希值。
c.params= hash	设置使用字段名和值。
c.password_field(name[, value[, size=40[, max]]]) c.password_field(options)	返回定义一个password字段的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.popup_menu(name, value...) c.popup_menu(options) c.scrolling_list(name, value...) c.scrolling_list(options)	返回定义一个弹出式菜单的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.radio_button(name[, value[, checked=false]]) c.radio_button(options)	返回定义一个radio字段的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.radio_group(name, value...) c.radio_group(options)	返回定义一个radio按钮组的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.reset(name[, value]) c.reset(options)	返回定义一个reset按钮的HTML字符串。 标签的属性可以以一个哈希函数作为参数传递
c.text_field(name[, value[, size=40[, max]]]) c.text_field(options)	返回定义一个text字段的HTML字符串。标签的属性可以以一个哈希函数作为参数传递。
c.textarea(name[, cols=70[,	返回定义一个textarea字段的HTML字符串。 如

rows=10]]) { ...} c.textarea(options) { ...}	textarea 的内容。 标签的属性可以以一个哈希函数作为参数传递。

HTML 生成方法

你可以再 CGI 实例中使用相应的 HTML 标签名来创建 HTML 标签，实例如下：

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cgi.out{
   cgi.html{
      cgi.head{ "\n"+cgi.title{"This Is a Test"} } +
      cgi.body{ "\n"+
         cgi.form{"\n"+
            cgi.hr +
            cgi.h1 { "A Form: " } + "\n"+
            cgi.textarea("get_text") +"\n"+
            cgi.br +
            cgi.submit
         }
      }
   }
}
```

CGI 对象属性

你可以再 CGI 实例中使用以下属性：

属性	返回值
accept	可接受的 MIME 类型
accept_charset	可接受的字符集
accept_encoding	可接受的编码
accept_language	可接受的语言
auth_type	可接受的类型
raw_cookie	Cookie 数据 (原字符串)
content_length	内容长度（Content length）
content_type	内容类型（Content type）
From	Client e-mail 地址
gateway_interface	CGI 版本
path_info	路径
path_translated	转换后的路径
Query_string	查询字符串
referer	之前访问网址
remote_addr	客户端主机地址(IP)
remote_host	客户端主机名
remote_ident	客户端名
remote_user	经过身份验证的用户
request_method	请求方法(GET, POST, 等。)
script_name	参数名
server_name	服务器名
server_port	服务器端口
server_protocol	服务器协议
server_software	服务器软件
user_agent	用户代理（User agent）

Ruby CGI Cookies

HTTP协议是无状态协议。但对于一个商业网站，它需要保持不同的页面间的会话信息。

如用户在网站注册过程中需要跳转页面，但又要保证之前填写的信息部丢失。

这种情况下 Cookie 很好的帮我们解决了问题。

Cookie 是如何工作的？

几乎所有的网站设计者在进行网站设计时都使用了Cookie，因为他们都想给浏览网站的用户提供一个更友好的、人文化的浏览环境，同时也能更加准确地收集访问者的信息。

写入和读取

Cookies集合是附属于Response对象及Request对象的数据集合，使用时需要在前面加上Response或Request。

用于给客户机发送Cookies的语法通常为：

当给不存在的Cookies集合设置时，就会在客户机创建，如果该Cookies己存在，则会被代替。由于Cookies是作为HTTP传输的头信息的一部分发给客户机的，所以向客户机发送Cookies的代码一般放在发送给浏览器的HTML文件的标记之前。

如果用户要读取Cookies，则必须使用Request对象的Cookies集合，其使用方法是： 需要注意的是，只有在服务器未被下载任何数据给浏览器前，浏览器才能与Server进行Cookies集合的数据交换，一旦浏览器开始接收Server所下载的数据，Cookies的数据交换则停止，为了避免错误，要在程序和前面加上response.Buffer=True。

集合的属性

- **1.Expires**属性：此属性用来给Cookies设置一个期限，在期限内只要打开网页就可以调用被保存的Cookies，如果过了此期限Cookies就自动被删除。如： 设定Cookies的有效期到2004年4月1日，到时将自动删除。如果一个Cookies没有设定有效期，则其生命周期从打开浏览器开始，到关闭浏览器结束，每次运行后生命周期将结束，下次运行将重新开始。
- **2.Domain**属性：这个属性定义了Cookies传送数据的唯一性。若只将某Cookies传送给_blank">搜狐主页时，则可使用如下代码：
- **3.Path**属性：定义了Cookies只发给指定的路径请求，如果Path属性没有被设置，则使用应用软件的缺省路径。
- **4.Secure**属性：指定Cookies能否被用户读取。

- **5、Name=Value :** Cookies是以键值对的形式进行设置和检索的。

Ruby 中处理Cookies

你可以创建一个名为 cookie 的对象并存储文本信息，将该信息发送至浏览器，调用 CGI.out 设置cookie的头部：

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cookie = CGI::Cookie.new('name' => 'mycookie',
                         'value' => 'Zara Ali',
                         'expires' => Time.now + 3600)
cgi.out('cookie' => cookie) do
   cgi.head + cgi.body { "Cookie stored" }
end
```

接下来我们回到这个页面，并查找cookie值，如下所示：

```
#!/usr/bin/ruby

require "cgi"
cgi = CGI.new("html4")
cookie = cgi.cookies['mycookie']
cgi.out('cookie' => cookie) do
   cgi.head + cgi.body { cookie[0] }
end
```

CGI::Cookie对象实例化时包含以下参数：

参数	描述
name	规定 cookie 的名称。
value	规定 cookie 的值。
expire	规定 cookie 的有效期。
path	规定 cookie 的服务器路径。
domain	规定 cookie 的域名。
secure	规定是否通过安全的 HTTPS 连接来传输 cookie。

Ruby CGI Sessions

CGI::Session 可以为用户和CGI环境保存持久的会话状态，会话使用后需要关闭，这样可以保证数据写入到存储当中，当会话完成后，你需要删除该数据。

```
#!/usr/bin/ruby

require 'cgi'
require 'cgi/session'
cgi = CGI.new("html4")

sess = CGI::Session.new( cgi, "session_key" => "a_test",
                              "prefix" => "rubysess.")
lastaccess = sess["lastaccess"].to_s
sess["lastaccess"] = Time.now
if cgi['bgcolor'][0] =~ /[a-z]/
  sess["bgcolor"] = cgi['bgcolor']
end

cgi.out{
  cgi.html {
    cgi.body ("bgcolor" => sess["bgcolor"]){
      "The background of this page"    +
      "changes based on the 'bgcolor'" +
      "each user has in session."      +
      "Last access time: #{lastaccess}"
    }
  }
}
```

访问 "/cgi-bin/test.cgi?bgcolor=red" 将跳转到指定背景颜色的页面。

会话数据存在在服务器的临时文件目录中，prefix 参数指定了会话的前缀，将作为临时文件的前缀。这样你在服务器上可以轻松的识别不同的会话临时文件。

CGI::Session 类

CGI::Session 保持了用户与 CGI 环境的持久状态。 会话可以在内存中，也可以在硬盘上。

类方法

Ruby 类 Class CGI::Session 提供了简单的方法来创建 session:

```
CGI::Session::new( cgi[, option])
```

启用一个新的 CGI 会话并返回相应的 CGI::Session 对象。选项可以是可选的哈希，可以是以下值：

- **session_key:** 键名保存会话 默认为 _session_id。

- **session_id:** 唯一的会话 ID。自动生成
- **new_session:** 如果为true，为当前会话创建一个新的Session id。 如果为 false, 通过 session_id 使用已存在的 session 标识。 如果省略该参数，如果可用则使用现有的会话，否则创建一个新的。
- **database_manager:** 用于保存 sessions 的类，可以是 CGI::Session::FileStore or CGI::Session::MemoryStore。默认为 FileStore。
- **tmpdir:** 对于 FileStore, 为 session 的错存储目录。
- **prefix:** 对于 FileStore, 为 session 文件的前缀。

实例化方法

方法	描述
[ ]	返回给定 key 的值。查看实例。
[ ]=	设置给定 key 的值。 查看实例。
delete	调用底层数据库管理的删除方法。对于 FileStore, 删除包含 session 的物理文件。 对于 MemoryStore, 从内存中移除 session 数据。
update	调用底层数据库管理的更新方法。 对于 FileStore, 将 session 写入到磁盘中。 对于 MemoryStore则无效果。

Ruby 发送邮件 - SMATP

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

Ruby提供了 Net::SMTP 来发送邮件，并提供了两个方法 new 和 start:

- **new** 方法有两个参数：
  - *server name* 默认为 localhost
  - *port number* 默认为 25
- **start** 方法有以下参数：
  - *server* - SMTP 服务器 IP, 默认为 localhost
  - *port* - 端口号，默认为 25
  - *domain* - 邮件发送者域名，默认为 ENV["HOSTNAME"]
  - *account* - 用户名，默认为 nil
  - *password* - 用户密码，默认为nil
  - *authtype* - 验证类型，默认为 *cram_md5*

SMTP 对象实例化方法调用了 sendmail, 参数如下：

- *source* - 一个字符串或数组或每个迭代器在任一时间中返回的任何东西。
- *sender* -一个字符串，出现在 email 的表单字段。
- *recipients* - 一个字符串或字符串数组，表示收件人的地址。

实例

以下提供了简单的Ruby脚本来发送邮件:

```
require 'net/smtp'

message = <<MESSAGE_END
From: Private Person <me@fromdomain.com>
To: A Test User <test@todomain.com>
Subject: SMTP e-mail test

This is a test e-mail message.
MESSAGE_END

Net::SMTP.start('localhost') do |smtp|
  smtp.send_message message, 'me@fromdomain.com',
                             'test@todomain.com'
end
```

在以上实例中，你已经设置了一个基本的电子邮件消息，注意正确的标题格式。一个电子邮件要要From，To和Subject，文本内容与头部信息间需要一个空行。

使用Net::SMTP连接到本地机器上的SMTP服务器，使用send_message方法来发送邮件，方法参数为发送者邮件与接收者邮件。

如果你没有运行在本机上的SMTP服务器，您可以使用Net::SMTP与远程SMTP服务器进行通信。如果使用网络邮件服务（如Hotmail或雅虎邮件），您的电子邮件提供者会为您提供发送邮件服务器的详细信息:

```
Net::SMTP.start('mail.your-domain.com')
```

以上代码将连接主机为 mail.your-domain.com，端口号为 25的邮件服务器，如果需要填写用户名密码，则代码如下：

```
Net::SMTP.start('mail.your-domain.com',
                25,
                'localhost',
                'username', 'password' :plain)
```

以上实例使用了指定的用户名密码连接到主机为 mail.your-domain.com，端口号为 25的邮件服务器。

使用 Ruby 发送 HTML 邮件

Net::SMTP同样提供了支持发送 HTML 格式的邮件。

发送电子邮件时你可以设置MIME版本，文档类型，字符集来发送HTML格式的邮件。

实例

以下实例用于发送 HTML 格式的邮件：

```
require 'net/smtp'

message = <<MESSAGE_END
From: Private Person <me@fromdomain.com>
To: A Test User <test@todomain.com>
MIME-Version: 1.0
Content-type: text/html
Subject: SMTP e-mail test

This is an e-mail message to be sent in HTML format

<b>This is HTML message.</b>
<h1>This is headline.</h1>
MESSAGE_END

Net::SMTP.start('localhost') do |smtp|
  smtp.send_message message, 'me@fromdomain.com',
                             'test@todomain.com'
end
```

发送带附件的邮件

如果需要发送混合内容的电子邮件，需要设置Content-type为multipart/mixed。 这样就可以在邮件中添加附件内容。

附件在传输前需要使用 **pack("m")** 函数将其内容转为 base64 格式。

实例

以下实例将发送附件为 /tmp/test.txt 的邮件：

```
require 'net/smtp'

filename = "/tmp/test.txt"
# 读取文件并编码为base64格式
filecontent = File.read(filename)
encodedcontent = [filecontent].pack("m")   # base64

marker = "AUNIQUEMARKER"

body =<<EOF
This is a test email to send an attachement.
EOF

# 定义主要的头部信息
part1 =<<EOF
From: Private Person <me@fromdomain.net>
To: A Test User <test@todmain.com>
Subject: Sending Attachement
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=#{marker}
--#{marker}
EOF

# 定义消息动作
part2 =<<EOF
Content-Type: text/plain
Content-Transfer-Encoding:8bit

#{body}
--#{marker}
EOF

# 定义附件部分
part3 =<<EOF
Content-Type: multipart/mixed; name=\"#{filename}\"
Content-Transfer-Encoding:base64
Content-Disposition: attachment; filename="#{filename}"

#{encodedcontent}
--#{marker}--
EOF

mailtext = part1 + part2 + part3

# 发送邮件
begin
  Net::SMTP.start('localhost') do |smtp|
     smtp.sendmail(mailtext, 'me@fromdomain.net',
                          ['test@todmain.com'])
  end
rescue Exception => e
  print "Exception occured: " + e
end
```

注意：你可以指定多个发送的地址，但需要使用逗号隔开。

Ruby Socket 编程

Ruby提供了两个级别访问网络的服务，在底层你可以访问操作系统，它可以让你实现客户端和服务器为面向连接和无连接协议的基本套接字支持。

Ruby 统一支持应用程的网络协议，如FTP、HTTP等。

不管是高层的还是底层的。ruby提供了一些基本类，让你可以使用TCP,UDP,SOCKS等很多协议交互，而不必拘泥在网络层。这些类也提供了辅助类，让你可以轻松的对服务器进行读写。

接下来就让我们来学习如何进行 Ruby Socket 编程

什么是 Sockets

应用层通过传输层进行数据通信时，TCP和UDP会遇到同时为多个应用程序进程提供并发服务的问题。多个TCP连接或多个应用程序进程可能需要 通过同一个TCP协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与TCP／IP协议交互提供了称为套接字 (Socket)的接口，区分不同应用程序进程间的网络通信和连接。

生成套接字，主要有3个参数：通信的目的IP地址、使用的传输 层协议(TCP或UDP)和使用的端口号。Socket原意是"插座"。通过将这3个参数结合起来，与一个"插座"Socket绑定，应用层就可以和传输 层通过套接字接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

Sockets 词汇解析：

选项	描述
domain	指明所使用的协议族，通常为 PF_INET, PF_UNIX, PF_X25, 等等。
type	指定socket的类型：SOCK_STREAM 或SOCK_DGRAM，Socket接口还定义了原始Socket（SOCK_RAW），允许程序使用低层协议
protocol	通常赋值0。
hostname	网络接口的标识符： 字符串, 可以是主机名或IP地址。字符串 " `<broadcast>` ", 指定 INADDR_BROADCAST 地址。 0 长度的字符串, 指定 `INADDR_ANY *` 。 一个整数，解释为主机字节顺序的二进制地址。
port	port是端口的编号，每个服务器都会监听客户端连接的一个或多个端口号，一个端口号可以是 Fixnum 的端口号, 包含了服务器名和端口。

简单的客户端

以下我们通过给定的主机和端口编写了一个简单的客户端实例，Ruby TCPSocket 类提供了 open 方法来打开一个 socke。

TCPSocket.open(hosname, port) 打开一个 TCP 连接。

一旦你打开一个 Socket 连接，你可以像 IO 对象一样读取它，完成后，你需要像关闭文件一样关闭该连接。

以下实例演示了如何连接到一个指定的主机，并从 socket 中读取数据，最后关闭socket：

```
require 'socket'      # Sockets 是标准库

hostname = 'localhost'
port = 2000

s = TCPSocket.open(hostname, port)

while line = s.gets   # 从 socket 中读取每行数据
  puts line.chop      # 打印到终端
end
s.close               # 关闭 socket
```

简单的服务

Ruby 中可以使用 TCPServer 类来写个简单的服务。TCPServer 对象是 TCPSocket 的工厂对象。

现在我们使用 TCPServer.open(hostname, port) 来创建一个 TCPServer 对象。

接下来调用 TCPServer 的 accept 方法，该方法会等到一个客户端连接到指定的端口，然后返回一个的TCPSocket对象，表示连接到该客户端。

```
require 'socket'                # 获取socket标准库

server = TCPServer.open(2000)  # Socket 监听端口为 2000
loop {                          # 永久运行服务
  client = server.accept        # 等待客户端连接
  client.puts(Time.now.ctime)   # 发送时间到客户端
  client.puts "Closing the connection. Bye!"
  client.close                  # 关闭客户端连接
}
```

现在，在服务器上运行以上代码，查看效果。

多客户端TCP服务

互联网上，大多服务都有大量的客户端连接。

Ruby的Thread类可以很容易地创建多线程服务，一个线程执行客户端的连接，而主线程在等待更多的连接。

```
require 'socket'                # 获取socket标准库

server = TCPServer.open(2000)   # Socket 监听端口为 2000
loop {                          # 永久运行服务
  Thread.start(server.accept) do |client|
    client.puts(Time.now.ctime) # 发送时间到客户端
    client.puts "Closing the connection. Bye!"
    client.close                # 关闭客户端连接
  end
}
```

在这个例子中，socket永久运行，而当server.accept接收到客户端的连接时，一个新的线程被创建并立即开始处理请求。而主程序立即循环回，并等待新的连接。

微小的Web浏览器

我们可以使用socket库来实现任何的 Internet 协议。以下代码展示了如何获取网页的内容：

```
require 'socket'

host = 'www.w3cschool.cc'     # web服务器
port = 80                           # 默认 HTTP 端口
path = "/index.htm"                 # 想要获取的文件地址

# 这是个 HTTP 请求
request = "GET #{path} HTTP/1.0\r\n\r\n"

socket = TCPSocket.open(host,port)  # 连接服务器
socket.print(request)               # 发送请求
response = socket.read              # 读取完整的响应
# Split response at first blank line into headers and body
headers,body = response.split("\r\n\r\n", 2)
print body                          # 输出结果
```

要实现一个类似 web 的客户端，你可以使用为 HTTP 预先构建的库如Net::HTTP。

以下代码与先前代码是等效的：

```
require 'net/http'                  # 我们需要的库
host = 'www.w3cschool.cc'           #  web 服务器
path = '/index.htm'                 # 我们想要的文件

http = Net::HTTP.new(host)          # 创建连接
headers, body = http.get(path)      # 请求文件
if headers.code == "200"            # 检测状态码
  print body
else
  puts "#{headers.code} #{headers.message}"
end
```

以上我们只是简单的为大家介绍 Ruby 中socket的应用，更多文档请查看：Ruby Socket 库和类方法

Ruby XML, XSLT 和 XPath 教程

什么是 XML ?

XML 指可扩展标记语言（eXtensible Markup Language）。

可扩展标记语言，标准通用标记语言的子集，一种用于标记电子文件使其具有结构性的标记语言。

它可以用来标记数据、定义数据类型，是一种允许用户对自己的标记语言进行定义的源语言。 它非常适合万维网传输，提供统一的方法来描述和交换独立于应用程序或供应商的结构化数据。

更多内容请查看我们的 XML 教程

XML解析器结构和API

XML的解析器主要有DOM和SAX两种。

- SAX解析器是基于事件处理的，需要从头到尾把XML文档扫描一遍，在扫描的过程中，每次遇到一个语法结构时，就会调用这个特定语法结构的事件处理程序，向应用程序发送一个事件。
- DOM是文档对象模型解析，构建文档的分层语法结构，在内存中建立DOM树，DOM树的节点以对象的形式来标识，文档解析文成以后，文档的整个DOM树都会放在内存中。

Ruby 中解析及创建 XML

RUBY中对XML的文档的解析可以使用这个库REXML库。

REXML库是ruby的一个XML工具包，是使用纯Ruby语言编写的，遵守XML1.0规范。

在Ruby1.8版本及其以后，RUBY标准库中将包含REXML。

REXML库的路径是： rexml/document

所有的方法和类都被封装到一个REXML模块内。

REXML解析器比其他的解析器有以下优点：

- 100% 由 Ruby 编写。
- 可适用于 SAX 和 DOM 解析器。
- 它是轻量级的,不到2000行代码。

- 很容易理解的方法和类。
- 基于 SAX2 API 和完整的 XPath 支持。
- 使用 Ruby 安装，而无需单独安装。

以下为实例的 XML 代码，保存为movies.xml:

```
<collection shelf="New Arrivals">
<movie title="Enemy Behind">
   <type>War, Thriller</type>
   <format>DVD</format>
   <year>2003</year>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
   <type>Anime, Science Fiction</type>
   <format>DVD</format>
   <year>1989</year>
   <rating>R</rating>
   <stars>8</stars>
   <description>A schientific fiction</description>
</movie>
   <movie title="Trigun">
   <type>Anime, Action</type>
   <format>DVD</format>
   <episodes>4</episodes>
   <rating>PG</rating>
   <stars>10</stars>
   <description>Vash the Stampede!</description>
</movie>
<movie title="Ishtar">
   <type>Comedy</type>
   <format>VHS</format>
   <rating>PG</rating>
   <stars>2</stars>
   <description>Viewable boredom</description>
</movie>
</collection>
```

DOM 解析器

让我们先来解析 XML 数据，首先我们先引入 rexml/document 库，通常我们可以将 REXML 在顶级的命名空间中引入：

```
#!/usr/bin/ruby -w

require 'rexml/document'
include REXML

xmlfile = File.new("movies.xml")
xmldoc = Document.new(xmlfile)

# 获取 root 元素
root = xmldoc.root
puts "Root element : " + root.attributes["shelf"]

# 以下将输出电影标题
xmldoc.elements.each("collection/movie"){
   |e| puts "Movie Title : " + e.attributes["title"]
}

# 以下将输出所有电影类型
xmldoc.elements.each("collection/movie/type") {
   |e| puts "Movie Type : " + e.text
}

# 以下将输出所有电影描述
xmldoc.elements.each("collection/movie/description") {
   |e| puts "Movie Description : " + e.text
}
```

以上实例输出结果为：

```
Root element : New Arrivals
Movie Title : Enemy Behind
Movie Title : Transformers
Movie Title : Trigun
Movie Title : Ishtar
Movie Type : War, Thriller
Movie Type : Anime, Science Fiction
Movie Type : Anime, Action
Movie Type : Comedy
Movie Description : Talk about a US-Japan war
Movie Description : A schientific fiction
Movie Description : Vash the Stampede!
Movie Description : Viewable boredom
SAX-like Parsing:
```

SAX 解析器

处理相同的数据文件：movies.xml，不建议SAX的解析为一个小文件，以下是个简单的实例：

```
#!/usr/bin/ruby -w

require 'rexml/document'
require 'rexml/streamlistener'
include REXML

class MyListener
  include REXML::StreamListener
  def tag_start(*args)
    puts "tag_start: #{args.map {|x| x.inspect}.join(', ')}"
  end

  def text(data)
    return if data =~ /^\w*$/     # whitespace only
    abbrev = data[0..40] + (data.length > 40 ? "..." : "")
    puts "  text   :   #{abbrev.inspect}"
  end
end

list = MyListener.new
xmlfile = File.new("movies.xml")
Document.parse_stream(xmlfile, list)
```

以上输出结果为：

```
tag_start: "collection", {"shelf"=>"New Arrivals"}
tag_start: "movie", {"title"=>"Enemy Behind"}
tag_start: "type", {}
  text   :   "War, Thriller"
tag_start: "format", {}
tag_start: "year", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text   :   "Talk about a US-Japan war"
tag_start: "movie", {"title"=>"Transformers"}
tag_start: "type", {}
  text   :   "Anime, Science Fiction"
tag_start: "format", {}
tag_start: "year", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text   :   "A schientific fiction"
tag_start: "movie", {"title"=>"Trigun"}
tag_start: "type", {}
  text   :   "Anime, Action"
tag_start: "format", {}
tag_start: "episodes", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text   :   "Vash the Stampede!"
tag_start: "movie", {"title"=>"Ishtar"}
tag_start: "type", {}
tag_start: "format", {}
tag_start: "rating", {}
tag_start: "stars", {}
tag_start: "description", {}
  text   :   "Viewable boredom"
```

XPath 和 Ruby

我们可以使用XPath来查看XML ,XPath 是一门在 XML 文档中查找信息的语言(查看：XPath 教程)。

XPath即为XML路径语言，它是一种用来确定XML（标准通用标记语言的子集）文档中某部分位置的语言。XPath基于XML的树状结构，提供在数据结构树中找寻节点的能力。

Ruby 通过 REXML 的 XPath 类支持 XPath，它是基于树的分析（文档对象模型）。

```
#!/usr/bin/ruby -w

require 'rexml/document'
include REXML

xmlfile = File.new("movies.xml")
xmldoc = Document.new(xmlfile)

# 第一个电影的信息
movie = XPath.first(xmldoc, "//movie")
p movie

# 打印所有电影类型
XPath.each(xmldoc, "//type") { |e| puts e.text }

# 获取所有电影格式的类型，返回数组
names = XPath.match(xmldoc, "//format").map {|x| x.text }
p names
```

以上实例输出结果为：

```
<movie title='Enemy Behind'> ... </>
War, Thriller
Anime, Science Fiction
Anime, Action
Comedy
["DVD", "DVD", "DVD", "VHS"]
```

XSLT 和 Ruby

Ruby 中有两个 XSLT 解析器，以下给出简要描述：

Ruby-Sablotron

这个解析器是由正义Masayoshi Takahash编写和维护。这主要是为Linux操作系统编写的，需要以下库：

- Sablot
- Iconv
- Expat

你可以在 Ruby-Sablotron 找到这些库。

XSLT4R

XSLT4R 由 Michael Neumann 编写。 XSLT4R 用于简单的命令行交互，可以被第三方应用程序用来转换XML文档。

XSLT4R需要XMLScan操作，包含了 XSLT4R 归档，它是一个100%的Ruby的模块。这些模块可以使用标准的Ruby安装方法（即Ruby install.rb）进行安装。

XSLT4R 语法格式如下：

```
ruby xslt.rb stylesheet.xsl document.xml [arguments]
```

如果您想在应用程序中使用XSLT4R，您可以引入XSLT及输入你所需要的参数。实例如下：

```
require "xslt"

stylesheet = File.readlines("stylesheet.xsl").to_s
xml_doc = File.readlines("document.xml").to_s
arguments = { 'image_dir' => '/....' }

sheet = XSLT::Stylesheet.new( stylesheet, arguments )

# output to StdOut
sheet.apply( xml_doc )

# output to 'str'
str = ""
sheet.output = [ str ]
sheet.apply( xml_doc )
```

更多资料

- 完整的 REXML 解析器, 请查看文档 REXML 解析器文档。
- 你可以从 RAA 知识库 中下载 XSLT4R 。

Ruby Web Services 应用 - SOAP4R

什么是 SOAP？

简单对象访问协议(SOAP,全写为Simple Object Access Protocol)是交换数据的一种协议规范。

SOAP 是一种简单的基于 XML 的协议，它使应用程序通过 HTTP 来交换信息。

简单对象访问协议是交换数据的一种协议规范，是一种轻量的、简单的、基于XML（标准通用标记语言下的一个子集）的协议，它被设计成在WEB上交换结构化的和固化的信息。

更多 SOAP 教程请查看：http://www.w3cschool.cc/soap/soap-tutorial.html。

SOAP4R 安装

SOAP4R 由Hiroshi Nakamura开发实现，用于 Ruby 的 SOAP 应用。

SOAP4R 下载地址：http://raa.ruby-lang.org/project/soap4r/。

注意：你的ruby环境可能已经安装了该该组件。

Linux 环境下你也可以使用 gem 来安装该组件，命令如下：

```
$ gem install soap4r --include-dependencies
```

如果你是window环境下开发，你需要下载zip压缩文件，并通过执行 install.rb 来安装。

SOAP4R 服务

SOAP4R 支持两种不同的服务类型：

- 基于 CGI/FastCGI 服务 (SOAP::RPC::CGIStub)
- 独立服务 (SOAP::RPC:StandaloneServer)

本教程将为大家介绍如何建立独立的 SOAP 服务。步骤如下：

第1步 - 继承SOAP::RPC::StandaloneServer

为了实现自己的独立的服务器，你需要编写一个新的类，该类为 SOAP::RPC::StandaloneServer 的子类：

```
class MyServer < SOAP::RPC::StandaloneServer
  ...............
end
```

注意：如果你要编写一个基于FastCGI的服务器，那么你需要继承 SOAP::RPC::CGIStub 类，程序的其余部分将保持不变。

第二步 - 定义处理方法

接下来我们定义Web Services的方法，如下我们定义两个方法，一个是两个数相加，一个是两个数相除：

```
class MyServer < SOAP::RPC::StandaloneServer
   ...............

   # 处理方法
   def add(a, b)
      return a + b
   end
   def div(a, b)
      return a / b
   end
end
```

第三步 - 公布处理方法

接下来添加我们在服务器上定义的方法，initialize方法是公开的，用于外部的连接：

```
class MyServer < SOAP::RPC::StandaloneServer
   def initialize(*args)
      add_method(receiver, methodName, *paramArg)
   end
end
```

以下是各参数的说明：

参数	描述
receiver	包含方法名的方法的对象。 如果你在同一个类中定义服务方法，该参数为 *self*。
methodName	调用 RPC 请求的方法名。
paramArg	参数名和参数模式

为了理解 *inout* 和 *out* 参数，考虑以下服务方法，需要输入两个参数:inParam 和 inoutParam，函数执行完成后返回三个值：retVal、inoutParam 、outParam:

```
def aMeth(inParam, inoutParam)
   retVal = inParam + inoutParam
   outParam = inParam . inoutParam
   inoutParam = inParam * inoutParam
   return retVal, inoutParam, outParam
end
```

公开的调用方法如下：

```
add_method(self, 'aMeth', [
    %w(in inParam),
    %w(inout inoutParam),
    %w(out outParam),
    %w(retval return)
])
```

第四步 - 开启服务

最后我们通过实例化派生类，并调用 start 方法来启动服务：

```
myServer = MyServer.new('ServerName',
                        'urn:ruby:ServiceName', hostname, port)

myServer.start
```

以下是请求参数的说明：

参数	描述
ServerName	服务名，你可以取你喜欢的
urn:ruby:ServiceName	Here *urn:ruby* 是固定的，但是你可以为你的服务取一个唯一的 *ServiceName*
hostname	指定主机名
port	web 服务端口

实例

接下来我们通过以上的步骤，创建一个独立的服务：

```
require "soap/rpc/standaloneserver"

begin
   class MyServer < SOAP::RPC::StandaloneServer

      # Expose our services
      def initialize(*args)
         add_method(self, 'add', 'a', 'b')
         add_method(self, 'div', 'a', 'b')
      end

      # Handler methods
      def add(a, b)
         return a + b
      end
      def div(a, b)
         return a / b
      end
  end
  server = MyServer.new("MyServer",
            'urn:ruby:calculation', 'localhost', 8080)
  trap('INT){
     server.shutdown
  }
  server.start
rescue => err
  puts err.message
end
```

执行以上程序后，就启动了一个监听 8080 端口的本地服务，并公开两个方法：add 和 div。

你可以再后台执行以上服务：

```
$ ruby MyServer.rb&
```

SOAP4R 客户端

ruby 中使用 SOAP::RPC::Driver 类开发 SOAP 客户端。接下来我们来详细看下 SOAP::RPC::Driver 类的使用。

调用 SOAP 服务需要以下信息：

- SOAP 服务 URL 地址 (SOAP Endpoint URL)
- 服务方法的命名空间（Method Namespace URI）
- 服务方法名及参数信息

接下来我们就一步步来创建 SOAP 客户端来调用以上的 SOAP 方法：add 、 div:

第一步 - 创建 SOAP Driver 实例

我们可以通过实例化 SOAP::RPC::Driver 类来调用它的新方法，如下所示：

```
SOAP::RPC::Driver.new(endPoint, nameSpace, soapAction)
```

以下是参数的描述：

参数	描述
endPoint	连接 SOAP 服务的 URL 地址
nameSpace	命名空间用于 SOAP::RPC::Driver 对象的所有 RPC .
soapAction	用于 HTTP 头部的 SOAPAction 字段值。如果是字符串是"" 则默认为 *nil*

第二步 - 添加服务方法

为 SOAP::RPC::Driver 添加 SOAP 服务方法，我们可以通过实例 SOAP::RPC::Driver 来调用以下方法：

```
driver.add_method(name, *paramArg)
```

以下是参数的说明：

参数	描述
name	远程web服务的方法名
paramArg	指定远程程序的参数

第三步 - 调用**SOAP**服务

最后我们可以使用 SOAP::RPC::Driver 实例来调用 SOAP 服务：

```
result = driver.serviceMethod(paramArg...)
```

serviceMethod SOAP服务的实际方法名，paramArg为方法的参数列表。

实例

基于以上的步骤，我们可以编写以下的 SOAP 客户端：

```
#!/usr/bin/ruby -w

require 'soap/rpc/driver'

NAMESPACE = 'urn:ruby:calculation'
URL = 'http://localhost:8080/'

begin
   driver = SOAP::RPC::Driver.new(URL, NAMESPACE)

   # Add remote sevice methods
   driver.add_method('add', 'a', 'b')

   # Call remote service methods
   puts driver.add(20, 30)
rescue => err
   puts err.message
end
```

以上我们只是简单介绍 Ruby 的 Web Services 。 如果你想了解更多可以查看官方文档：
Ruby 的 Web Services

Ruby 多线程

每个正在系统上运行的程序都是一个进程。每个进程包含一到多个线程。

线程是程序中一个单一的顺序控制流程，在单个程序中同时运行多个线程完成不同的工作,称为多线程。

Ruby 中我们可以通过 Thread 类来创建多线程，Ruby的线程是一个轻量级的，可以以高效的方式来实现并行的代码。

创建 Ruby 线程

要启动一个新的线程，只需要调用 Thread.new 即可:

```
# 线程 #1 代码部分
Thread.new {
  # 线程 #2 执行代码
}
# 线程 #1 执行代码
```

实例

以下实例展示了如何在Ruby程序中使用多线程：

```
#!/usr/bin/ruby

def func1
   i=0
   while i<=2
      puts "func1 at: #{Time.now}"
      sleep(2)
      i=i+1
   end
end

def func2
   j=0
   while j<=2
      puts "func2 at: #{Time.now}"
      sleep(1)
      j=j+1
   end
end

puts "Started At #{Time.now}"
t1=Thread.new{func1()}
t2=Thread.new{func2()}
t1.join
t2.join
puts "End at #{Time.now}"
```

以上代码执行结果为：

```
Started At Wed May 14 08:21:54 -0700 2014
func1 at: Wed May 14 08:21:54 -0700 2014
func2 at: Wed May 14 08:21:54 -0700 2014
func2 at: Wed May 14 08:21:55 -0700 2014
func1 at: Wed May 14 08:21:56 -0700 2014
func2 at: Wed May 14 08:21:56 -0700 2014
func1 at: Wed May 14 08:21:58 -0700 2014
End at Wed May 14 08:22:00 -0700 2014
```

线程生命周期

1、线程的创建可以使用Thread.new,同样可以以同样的语法使用Thread.start 或者Thread.fork 这三个方法来创建线程。

2、创建线程后无需启动，线程会自动执行。

3、Thread 类定义了一些方法来操控线程。线程执行Thread.new中的代码块。

4、线程代码块中最后一个语句是线程的值，可以通过线程的方法来调用，如果线程执行完毕，则返回线程值，否则不返回值直到线程执行完毕。

5、Thread.current 方法返回表示当前线程的对象。 Thread.main 方法返回主线程。

6、通过 Thread.Join 方法来执行线程，这个方法会挂起主线程，直到当前线程执行完毕。

线程状态

线程有5种状态：

线程状态	返回值
Runnable	run
Sleeping	Sleeping
Aborting	aborting
Terminated normally	false
Terminated with exception	nil

线程和异常

当某线程发生异常，且没有被rescue捕捉到时，该线程通常会被无警告地终止。但是，若有其它线程因为Thread#join的关系一直等待该线程的话，则等待的线程同样会被引发相同的异常。

```
begin
  t = Thread.new do
    Thread.pass    # 主线程确实在等join
    raise "unhandled exception"
  end
  t.join
rescue
  p $!  # => "unhandled exception"
end
```

使用下列3个方法，就可以让解释器在某个线程因异常而终止时中断运行。

- 启动脚本时指定**-d**选项，并以调试模时运行。
- 用 `Thread.abort_on_exception` 设置标志。
- 使用 `Thread#abort_on_exception` 对指定的线程设定标志。

当使用上述3种方法之一后，整个解释器就会被中断。

```
t = Thread.new { ... }
t.abort_on_exception = true
```

线程同步控制

在Ruby中，提供三种实现同步的方式，分别是：

1. 通过Mutex类实现线程同步

2. 监管数据交接的Queue类实现线程同步

3. 使用ConditionVariable实现同步控制

通过**Mutex**类实现线程同步

通过Mutex类实现线程同步控制，如果在多个线程钟同时需要一个程序变量，可以将这个变量部分使用lock锁定。 代码如下：

```
#encoding:gbk
require "thread"
puts "Synchronize Thread"

@num=200
@mutex=Mutex.new

def buyTicket(num)
    @mutex.lock
        if @num>=num
            @num=@num-num
            puts "you have successfully bought #{num} tickets"
        else
            puts "sorry,no enough tickets"
        end
    @mutex.unlock
end

ticket1=Thread.new 10 do
    10.times do |value|
    ticketNum=15
    buyTicket(ticketNum)
    sleep 0.01
    end
end

ticket2=Thread.new 10 do
    10.times do |value|
    ticketNum=20
    buyTicket(ticketNum)
    sleep 0.01
    end
end

sleep 1
ticket1.join
ticket2.join
```

输出结果如下：

```
Synchronize Thread
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
you have successfully bought 20 tickets
you have successfully bought 15 tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
sorry,no enough tickets
```

除了使用lock锁定变量，还可以使用try_lock锁定变量，还可以使用Mutex.synchronize同步对某一个变量的访问。

监管数据交接的Queue类实现线程同步

Queue类就是表示一个支持线程的队列，能够同步对队列末尾进行访问。不同的线程可以使用统一个对类，但是不用担心这个队列中的数据是否能够同步，另外使用SizedQueue类能够限制队列的长度

SizedQueue类能够非常便捷的帮助我们开发线程同步的应用程序，应为只要加入到这个队列中，就不用关心线程的同步问题。

经典的生产者消费者问题：

```
#encoding:gbk
require "thread"
puts "SizedQuee Test"

queue = Queue.new

producer = Thread.new do
    10.times do |i|
        sleep rand(i) # 让线程睡眠一段时间
        queue << i
        puts "#{i} produced"
    end
end

consumer = Thread.new do
    10.times do |i|
        value = queue.pop
        sleep rand(i/2)
        puts "consumed #{value}"
    end
end

consumer.join
```

程序的输出：

```
SizedQuee Test
0 produced
1 produced
consumed 0
2 produced
consumed 1
consumed 2
3 produced
consumed 34 produced

consumed 4
5 produced
consumed 5
6 produced
consumed 6
7 produced
consumed 7
8 produced
9 produced
consumed 8
consumed 9
```

使用ConditionVariable实现同步控制

使用 CondititonVariable进行同步控制，能够在一些致命的资源竞争部分挂起线程直到有可用的资源为止。

```
#encoding:gbk
require "thread"
puts "thread synchronize by ConditionVariable"

mutex = Mutex.new
resource = ConditionVariable.new

a = Thread.new {
    mutex.synchronize {
        # 这个线程目前需要resource这个资源
        resource.wait(mutex)
        puts "get resource"
    }
}

b = Thread.new {
    mutex.synchronize {
        #线程b完成对resourece资源的使用并释放resource
        resource.signal
    }
}

a.join
puts "complete"
```

mutex 是声明的一个资源，然后通过ConditionVariable来控制申请和释放这个资源。

b 线程完成了某些工作之后释放资源resource.signal,这样a线程就可以获得一个mutex资源然后进行执行。 执行结果：

```
thread synchronize by ConditionVariable
get resource
complete
```

线程类方法

完整的 Thread（线程） 类方法如下：

方法	描述
Thread.abort_on_exception	若其值为真的话，一旦某线程因异常而终止时，整个解释器就会被中断。它的默认值是假，也就是说，在通常情况下，若某线程发生异常且该异常未被Thread#join等检测到时，该线程会被无警告地终止。
Thread.abort_on_exception=	如果设置为 *true*, 一旦某线程因异常而终止时，整个解释器就会被中断。返回新的状态
Thread.critical	返回布尔值。
Thread.critical=	当其值为true时，将不会进行线程切换。若当前线程挂起(stop)或有信号(signal)干预时，其值将自动变为false。
Thread.current	返回当前运行中的线程(当前线程)。
Thread.exit	终止当前线程的运行。返回当前线程。若当前线程是唯一的一个线程时，将使用exit(0)来终止它的运行。
Thread.fork { block }	与 Thread.new 一样生成线程。
Thread.kill(aThread)	终止线程的运行.
Thread.list	返回处于运行状态或挂起状态的活线程的数组。
Thread.main	返回主线程。
Thread.new([arg]*) {| args | block }	生成线程,并开始执行。数会被原封不动地传递给块. 这就可以在启动线程的同时,将值传递给该线程所固有的局部变量。
Thread.pass	将运行权交给其他线程. 它不会改变运行中的线程的状态,而是将控制权交给其他可运行的线程(显式的线程调度)。
Thread.start([args]*) {| args | block }	生成线程,并开始执行。数会被原封不动地传递给块. 这就可以在启动线程的同时,将值传递给该线程所固有的局部变量。
Thread.stop	将当前线程挂起,直到其他线程使用run方法再次唤醒该线程。

线程实例化方法

以下实例调用了线程实例化方法 join：

```
#!/usr/bin/ruby

thr = Thread.new do   # 实例化
   puts "In second thread"
   raise "Raise exception"
end
thr.join   # 调用实例化方法 join
```

以下是完整实例化方法列表：

序号	方法描述
thr[name]	取出线程内与name相对应的固有数据。 name可以是字符串或符号。 若没有与name相对应的数据时, 返回nil。
thr[name] =	设置线程内name相对应的固有数据的值， name可以是字符串或符号。 若设为nil时, 将删除该线程内对应数据。
thr.abort_on_exception	返回布尔值。
thr.abort_on_exception=	若其值为true的话，一旦某线程因异常而终止时，整个解释器就会被中断。
thr.alive?	若线程是"活"的,就返回true。
thr.exit	终止线程的运行。返回self。
thr.join	挂起当前线程,直到self线程终止运行为止. 若self因异常而终止时, 将会当前线程引发同样的异常。
thr.key?	若与name相对应的线程固有数据已经被定义的话,就返回true
thr.kill	类似于 *Thread.exit* 。
thr.priority	返回线程的优先度. 优先度的默认值为0. 该值越大则优先度越高.
thr.priority=	设定线程的优先度. 也可以将其设定为负数.
thr.raise(anException)	在该线程内强行引发异常.
thr.run	重新启动被挂起(stop)的线程. 与wakeup不同的是,它将立即进行线程的切换. 若对死进程使用该方法时, 将引发ThreadError异常.
thr.safe_level	返回self 的安全等级. 当前线程的safe_level与$SAFE相同.
thr.status	使用字符串"run"、"sleep"或"aborting" 来表示活线程的状态. 若某线程是正常终止的话,就返回false. 若因异常而终止的话,就返回nil。
thr.stop?	若线程处于终止状态(dead)或被挂起(stop)时,返回true.
thr.value	一直等到self线程终止运行(等同于join)后,返回该线程的块的返回值. 若在线程的运行过程中发生了异常, 就会再次引发该异常.
thr.wakeup	把被挂起(stop)的线程的状态改为可执行状态(run), 若对死线程执行该方法时,将会引发ThreadError异常。

免责声明

W3School提供的内容仅用于培训。我们不保证内容的正确性。通过使用本站内容随之而来的风险与本站无关。W3School简体中文版的所有内容仅供测试，对任何法律问题及风险不承担任何责任。